华章科技

数据库技术丛书

Redis设计与实现

黄健宏 著

The Design and Implementation of Redis

- 系统而全面地描述了 Redis 内部运行机制。
- 图示丰富，描述清晰，并给出大量参考信息，是NoSQL数据库开发人员案头必备。
- 包括大部分Redis单机特征，以及所有多机特性。



机械工业出版社
China Machine Press

VISIT...

LANZAROTE

Caliente.COM

华章科技

数据库技术丛书

Redis设计与实现

黄健宏 著

The Design and Implementation of Redis

- 系统而全面地描述了 Redis 内部运行机制。
- 图示丰富，描述清晰，并给出大量参考信息，是NoSQL数据库开发人员案头必备。
- 包括大部分Redis单机特征，以及所有多机特性。



机械工业出版社
China Machine Press

数据库技术丛书

Redis设计与实现

黄健宏　著

ISBN：978-7-111-46474-7

本书纸版由机械工业出版社于2014年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：＋ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目录

前言
致谢
第1章 引言
- 1.1 Redis版本说明
- 1.2 章节编排
- 1.3 推荐的阅读方法
- 1.4 行文规则
- 1.5 配套网站

第一部分 数据结构与对象
- 第2章 简单动态字符串
- 2.1 SDS的定义
- 2.2 SDS与C字符串的区别
- 2.3 SDS API
- 2.4 重点回顾
- 2.5 参考资料
- 第3章 链表
- 3.1 链表和链表节点的实现
- 3.2 链表和链表节点的API
- 3.3 重点回顾
- 第4章 字典
- 4.1 字典的实现
- 4.2 哈希算法
- 4.3 解决键冲突
- 4.4 rehash
- 4.5 渐进式rehash
- 4.6 字典API
- 4.7 重点回顾
- 第5章 跳跃表
- 5.1 跳跃表的实现
- 5.2 跳跃表API
- 5.3 重点回顾
- 第6章 整数集合
- 6.1 整数集合的实现
- 6.2 升级
- 6.3 升级的好处
- 6.4 降级
- 6.5 整数集合API
- 6.6 重点回顾
- 第7章 压缩列表
- 7.1 压缩列表的构成
- 7.2 压缩列表节点的构成
- 7.3 连锁更新

7.4 压缩列表API
7.5 重点回顾
第8章 对象
8.1 对象的类型与编码
8.2 字符串对象
8.3 列表对象
8.4 哈希对象
8.5 集合对象
8.6 有序集合对象
8.7 类型检查与命令多态
8.8 内存回收
8.9 对象共享
8.10 对象的空转时长
8.11 重点回顾
第二部分 单机数据库的实现
第9章 数据库
9.1 服务器中的数据库
9.2 切换数据库
9.3 数据库键空间
9.4 设置键的生存时间或过期时间
9.5 过期键删除策略
9.6 Redis的过期键删除策略
9.7 AOF、RDB和复制功能对过期键的处理
9.8 数据库通知
9.9 重点回顾
第10章 RDB持久化
10.1 RDB文件的创建与载入
10.2 自动间隔性保存
10.3 RDB文件结构
10.4 分析RDB文件
10.5 重点回顾
10.6 参考资料
第11章 AOF持久化
11.1 AOF持久化的实现
11.2 AOF文件的载入与数据还原
11.3 AOF重写
11.4 重点回顾
第12章 事件
12.1 文件事件
12.2 时间事件
12.3 事件的调度与执行
12.4 重点回顾
12.5 参考资料
第13章 客户端

13.1 客户端属性
13.2 客户端的创建与关闭
13.3 重点回顾
第14章 服务器
14.1 命令请求的执行过程
14.2 serverCron函数
14.3 初始化服务器
14.4 重点回顾
第三部分 多机数据库的实现
第15章 复制
15.1 旧版复制功能的实现
15.2 旧版复制功能的缺陷
15.3 新版复制功能的实现
15.4 部分重同步的实现
15.5 PSYNC命令的实现
15.6 复制的实现
15.7 心跳检测
15.8 重点回顾
第16章 Sentinel
16.1 启动并初始化Sentinel
16.2 获取主服务器信息
16.3 获取从服务器信息
16.4 向主服务器和从服务器发送信息
16.5 接收来自主服务器和从服务器的频道信息
16.6 检测主观下线状态
16.7 检查客观下线状态
16.8 选举领头Sentinel
16.9 故障转移
16.10 重点回顾
16.11 参考资料
第17章 集群
17.1 节点
17.2 槽指派
17.3 在集群中执行命令
17.4 重新分片
17.5 ASK错误
17.6 复制与故障转移
17.7 消息
17.8 重点回顾
第四部分 独立功能的实现
第18章 发布与订阅
18.1 频道的订阅与退订
18.2 模式的订阅与退订
18.3 发送消息

18.4 查看订阅信息
18.5 重点回顾
18.6 参考资料
第19章 事务
19.1 事务的实现
19.2 WATCH命令的实现
19.3 事务的ACID性质
19.4 重点回顾
19.5 参考资料
第20章 Lua脚本
20.1 创建并修改Lua环境
20.2 Lua环境协作组件
20.3 EVAL命令的实现
20.4 EVALSHA命令的实现
20.5 脚本管理命令的实现
20.6 脚本复制
20.7 重点回顾
20.8 参考资料
第21章 排序
21.1 SORT命令的实现
21.2 ALPHA选项的实现
21.3 ASC选项和DESC选项的实现
21.4 BY选项的实现
21.5 带有ALPHA选项的BY选项的实现
21.6 LIMIT选项的实现
21.7 GET选项的实现
21.8 STORE选项的实现
21.9 多个选项的执行顺序
21.10 重点回顾
第22章 二进制位数组
22.1 位数组的表示
22.2 GETBIT命令的实现
22.3 SETBIT命令的实现
22.4 BITCOUNT命令的实现
22.5 BITOP命令的实现
22.6 重点回顾
22.7 参考资料
第23章 慢查询日志
23.1 慢查询记录的保存
23.2 慢查询日志的阅览和删除
23.3 添加新日志
23.4 重点回顾
第24章 监视器
24.1 成为监视器

24.2 向监视器发送命令信息
24.3 重点回顾

前言

时间回到2011年4月，当时我正在编写一个用户关系模块，这个模块需要实现一个“共同关注”功能，用于计算出两个用户关注了哪些相同的用户。

举个例子，假设huangz关注了peter、tom、jack三个用户，而john关注了peter、tom、bob、david四个用户，那么当huangz访问john的页面时，共同关注功能就会计算并打印出类似“你跟john都关注了peter和tom”这样的信息。

从集合计算的角度来看，共同关注功能本质上就是计算两个用户关注集合的交集，因为交集这个概念是如此的常见，所以我很自然地认为共同关注这个功能可以很容易地实现，但现实却给了我当头一棒：我所使用的关系数据库并不直接支持交集计算操作，要计算两个集合的交集，除了需要对两个数据表执行合并（join）操作之外，还需要对合并的结果执行去重复（distinct）操作，最终导致交集操作的实现变得异常复杂。

是否存在直接支持集合操作的数据库呢？带着这个疑问，我在搜索引擎上面进行查找，并最终发现了Redis。在我看来，Redis正是我想要找的那种数据库——它内置了集合数据类型，并支持对集合执行交集、并集、差集等集合计算操作，其中的交集计算操作可以直接用于实现我想要的共同关注功能。

得益于Redis本身的简单性，以及Redis手册的详尽和完善，我很快学会了怎样使用Redis的集合数据类型，并用它重新实现了整个用户关系模块：重写之后的关系模块不仅代码量更少，速度更快，更重要的是，之前需要使用一段甚至一大段SQL查询才能实现的功能，现在只需要调用一两个Redis命令就能够实现了，整个模块的可读性得到了极大的提高。

自此之后，我开始在越来越多的项目里面使用Redis，与此同时，我对Redis的内部实现也越来越感兴趣，一些问题开始频繁地出现在我的脑海中，比如：

·Redis的五种数据类型分别是由什么数据结构实现的？

·Redis的字符串数据类型既可以存储字符串（比如“hello world”），又可以存储整数和浮点数（比如10086和3.14），甚至是二进制位（使用SETBIT等命令），Redis在内部是怎样存储这些值的？

·Redis的一部分命令只能对特定数据类型执行（比如APPEND只能对字符串执行，HSET只能对哈希表执行），而另一部分命令却可以对所有

数据类型执行（比如DEL、TYPE和EXPIRE），不同的命令在执行时是如何进行类型检查的？Redis在内部是否实现了一个类型系统？

·Redis的数据库是怎样存储各种不同数据类型的键值对的？数据库里面的过期键又是怎样实现自动删除的？

·除了数据库之外，Redis还拥有发布与订阅、脚本、事务等特性，这些特性又是如何实现的？

·Redis使用什么模型或者模式来处理客户端的命令请求？一条命令请求从发送到返回需要经过什么步骤？

为了找到这些问题的答案，我再次在搜索引擎上面进行查找，可惜的是这次搜索并没有多少收获：Redis还是一个非常年轻的软件，对它的最好介绍就是官方网站上面的文档，但是这些文档主要关注的是怎样使用Redis，而不是介绍Redis的内部实现。另外，网上虽然有一些博客文章对Redis的内部实现进行了介绍，但这些文章要么不齐全（只介绍了Redis中的少数几个特性），要么就写得过于简单（只是一些概述性的文章），要么关注的就是旧版本（比如2.0、2.2或者2.4，而当时的最新版已经是2.6了）。

综合来看，详细而且完整地介绍Redis内部实现的资料，无论是外文还是中文都不存在。意识到这一点之后，我决定自己动手注释Redis的源代码，从中寻找问题的答案，并通过写博客的方式与其他Redis用户分享我的发现。在积累了七八篇Redis源代码注释文章之后，我想如果能将这些博文汇集成书的话，那一定会非常有趣，并且我自己也会从中学到很多知识。于是我在2012年年末开始创作《Redis设计与实现》，并最终于2013年3月8日在互联网发布了本书的第一版。

尽管《Redis设计与实现》第一版顺利发布了，但在我的心目中，这个第一版还是有很多不完善的地方：

·比如说，因为第一版是我边注释Redis源代码边写的，如果有足够时间让我先完整地注释一遍Redis的源代码，然后再进行写作的话，那么书本在内容方面应该会更为全面。

·又比如说，第一版只介绍了Redis的内部机制和单机特性，但并没有介绍Redis多机特性，而我认为只有将关于多机特性的介绍也包含进来，这本《Redis设计与实现》才算是真正的完成了。

就在我考虑应该何时编写新版来修复这些缺陷的时候，机械工业出版社的吴怡编辑来信询问我是否有兴趣正式地出版《Redis设计与实现》，能够正式地出版自己写的书一直是我梦寐以求的事情，我找不到任何拒绝这一邀请的理由，就这样，在《Redis设计与实现》第一版发布几天之后，新版《Redis设计与实现》的写作也马不停蹄地开始了。

从2013年3月到2014年1月这11个月间，我重新注释了Redis在unstable分支的源代码（也即是现在的Redis 3.0源代码），重写了《Redis设计与实现》第一版已有的所有章节，并向书中添加了关于二进制位操作（bitop）、排序、复制、Sentinel和集群等主题的新章节，最终完成了这本新版的《Redis设计与实现》。本书不仅介绍了Redis的内部机制（比如数据库实现、类型系统、事件模型），而且还介绍了大部分Redis单机特性（比如事务、持久化、Lua脚本、排序、二进制位操作），以及所有Redis多机特性（如复制、Sentinel和集群）。

虽然作者创作本书的初衷只是为了满足自己的好奇心，但了解Redis内部实现的好处并不仅仅在于满足好奇心：通过了解Redis的内部实现，理解每一个特性和命令背后的运作机制，可以帮助我们更高效地使用Redis，避开那些可能会引起性能问题的陷阱。我衷心希望这本新版《Redis设计与实现》能够帮助读者更好地了解Redis，并成为更优秀的Redis使用者。

本书的第一版获得了很多热心读者的反馈，这本新版的很多改进也来源于读者们的意见和建议，因此我将继续在www.RedisBook.com设置disqus论坛（可以不注册直接发贴），欢迎读者随时就这本新版《Redis设计与实现》发表提问、意见、建议、批评、勘误，等等，我会努力地采纳大家的意见，争取在将来写出更好的《Redis设计与实现》，以此来回报大家对本书的支持。

黄健宏（huangz）

2014年3月于清远

致谢

我要感谢hoterran和iammutex这两位良师益友，他们对我的帮助和支持贯穿整本书从概念萌芽到正式出版的整个阶段，也感谢他们抽出宝贵的时间为本书审稿。

我要感谢吴怡编辑鼓励我创作并出版这本新版《Redis设计与实现》，以及她在写作过程中对我的悉心指导。

我要感谢TimYang在百忙之中抽空为本书审稿，并耐心地给出了详细的意见。

我要感谢Redis之父Salvatore Sanfilippo，如果不是他创造了Redis的话，这本书也不会出现了。

我要感谢所有阅读了《Redis设计与实现》第一版的读者，他们的意见和建议帮助我更好地完成这本新版《Redis设计与实现》。

最后，我要感谢我的家人和朋友，他们的关怀和鼓励使得本书得以顺利完成。

第1章　引言

本书对Redis的大多数单机功能以及所有多机功能的实现原理进行了介绍，力图展示这些功能的核心数据结构以及关键的算法思想。

通过阅读本书，读者可以快速、有效地了解Redis的内部构造以及运作机制，这些知识可以帮助读者更好地、也更高效地使用Redis。

为了让本书的内容保持简单并且容易读懂，本书会尽量以高层次的角度来对Redis的实现原理进行描述，如果读者只是对Redis的实现原理感兴趣，但并不想研究Redis的源代码，那么阅读本书就足够了。

另一方面，如果读者打算深入了解Redis实现原理的底层细节，本书在RedisBook.com提供了一份带有详细注释的Redis源代码，读者可以先阅读本书对某一功能的介绍，然后再阅读该功能对应的实现代码，这有助于读者更快地读懂实现代码，也有助于读者更深入地了解该功能的实现原理。

1.1 Redis版本说明

本书是基于Redis 2.9——也即是Redis 3.0的开发版来编写的，因为Redis 3.0的更新主要与Redis的多机功能有关，而Redis 3.0的单机功能则与Redis 2.6、Redis 2.8的单机功能基本相同，所以本书的内容对于使用Redis 2.6至Redis 3.0的读者来说应该都是有用的。

另外，因为Redis通常都是渐进地增加新功能，并且很少会大幅地修改已有的功能，所以本书的大部分内容对于Redis 3.0之后的几个版本来说，应该也是有用的。

1.2 章节编排

本书由“数据结构与对象”、“单机数据库的实现”、“多机数据库的实现”、“独立功能的实现”四个部分组成。

第一部分“数据结构与对象”

Redis数据库里面的每个键值对（key-value pair）都是由对象（object）组成的，其中：

·数据库键总是一个字符串对象（string object）；

·而数据库键的值则可以是字符串对象、列表对象（list object）、哈希对象（hash object）、集合对象（set object）、有序集合对象（sorted set object）这五种对象中的其中一种。

比如说，执行以下命令将在数据库中创建一个键为字符串对象，值也为字符串对象的键值对：

```
redis> SET msg "hello world"
OK
```

而执行以下命令将在数据库中创建一个键为字符串对象，值为列表对象的键值对：

```
redis> RPUSH numbers 1 3 5 7 9
(integer) 5
```

本书的第一部分将对以上提到的五种不同类型的对象进行介绍，剖析这些对象所使用的底层数据结构，并说明这些数据结构是如何深刻地影响对象的功能和性能的。

第二部分“单机数据库的实现”

本书的第二部分对Redis实现单机数据库的方法进行了介绍。

第9章“数据库”对Redis数据库的实现原理进行了介绍，说明了服务器保存键值对的方法，服务器保存键值对过期时间的方法，以及服务器自动删除过期键值对的方法等等。

第10章“RDB持久化”和第11章“AOF持久化”分别介绍了Redis两种不同的持久化方式的实现原理，说明了服务器根据数据库来生成持久化文件的方法，服务器根据持久化文件来还原数据库的方法，以及BGSAVE命令和BGREWRITEAOF命令的实现原理等等。

第12章“事件”对Redis的文件事件和时间事件进行了介绍：

·文件事件主要用于应答（accept）客户端的连接请求，接收客户端发送的命令请求，以及向客户端返回命令回复；

·而时间事件则主要用于执行redis.c/serverCron函数，这个函数通过执行常规的维护和管理操作来保持Redis服务器的正常运作，一些重要的定时操作也是由这个函数负责触发的。

第13章“客户端”对Redis服务器维护和管理客户端状态的方法进行了介绍，列举了客户端状态包含的各个属性，说明了客户端的输入缓冲区和输出缓冲区的实现方法，以及Redis服务器创建和销毁客户端状态的条件等等。

第14章“服务器”对单机Redis服务器的运作机制进行了介绍，详细地说明了服务器处理命令请求的步骤，解释了serverCron函数所做的工作，并讲解了Redis服务器的初始化过程。

第三部分“多机数据库的实现”

本书的第三部分对Redis的Sentinel、复制（replication）、集群（cluster）三个多机功能进行了介绍。

第15章“复制”对Redis的主从复制功能（master-slave replication）的实现原理进行了介绍，说明了当用户指定一个服务器（从服务器）去复制另一个服务器（主服务器）时，主从服务器之间执行了什么操作，进行了什么数据交互，诸如此类。

第16章“Sentinel”对Redis Sentinel的实现原理进行了介绍，说明了Sentinel监视服务器的方法，Sentinel判断服务器是否下线的方法，以及Sentinel对下线服务器进行故障转移的方法等等。

第17章“集群”对Redis集群的实现原理进行了介绍，说明了节点（node）的构建方法，节点处理命令请求的方法，转发（redirection）错误的实现方法，以及各个节点之间进行通信的方法等等。

第四部分“独立功能的实现”

本书的第四部分对Redis中各个相对独立的功能模块进行了介绍。

第18章“发布与订阅”对PUBLISH、SUBSCRIBE、PUBSUB等命令的实现原理进行了介绍，解释了Redis的发布与订阅功能是如何实现的。

第19章“事务”对MULTI、EXEC、WATCH等命令的实现原理进行了介绍，解释了Redis的事务是如何实现的，并说明了Redis的事务对ACID性质的支持程度。

第20章“Lua脚本”对EVAL、EVALSHA、SCRIPT LOAD等命令的实

现原理进行了介绍，解释了Redis服务器是如何执行和管理用户传入的Lua脚本的；这一章还对Redis服务器构建Lua环境的过程，以及主从服务器之间复制Lua脚本的方法进行了介绍。

第21章“排序”对SORT命令以及SORT命令所有可用选项（比如DESC、ALPHA、GET等等）的实现原理进行了介绍，并说明了当SORT命令带有多个选项时，不同选项执行的先后顺序。

第22章“二进制位数组”对Redis保存二进制位数组的方法进行了介绍，并说明了GETBIT、SETBIT、BITCOUNT、BITOP这几个二进制位数组操作命令的实现原理。

第23章“慢查询日志”对Redis创建和保存慢查询日志（slow log）的方法进行了介绍，并说明了SLOWLOG GET、SLOWLOG LEN、SLOWLOG RESET等慢查询日志操作命令的实现原理。

第24章“监视器”介绍了将客户端变为监视器（monitor）的方法，以及服务器在处理命令请求时，向监视器发送命令信息的方法。

1.3 推荐的阅读方法

因为Redis的单机功能是多机功能的子集，所以无论读者使用的是单机模式的Redis，还是多机模式的Redis，都应该阅读本书的第一部分和第二部分，这两个部分包含的知识是所有Redis使用者都必然会用到的。

如果读者要使用Redis的多机功能，那么在阅读本书的第一部分和第二部分之后，应该接着阅读本书的第三部分。如果读者只使用Redis的单机功能，那么可以跳过第三部分，直接阅读第四部分。

本书的前三个部分都是以自底向上（bottom-up）的方式来写的，也就是说，排在后面的章节会假设读者已经读过了排在前面的章节。如果一个概念在前面的章节已经介绍过，那么后面的章节就不会再重复介绍这个概念，所以读者最好按顺序阅读这三部分的各个章节。

本书的第四部分包含的各章是完全独立的，读者可以按自己的兴趣来挑选要读的章节。在本书的第四部分中，除了第20章的其中一节涉及多机功能的内容之外，其他章节都没有涉及多机功能的内容，所以第四部分的大部分章节都可以在只阅读了本书第一部分和第二部分的情况下阅读。

图1-1对上面描述的阅读方法进行了总结。

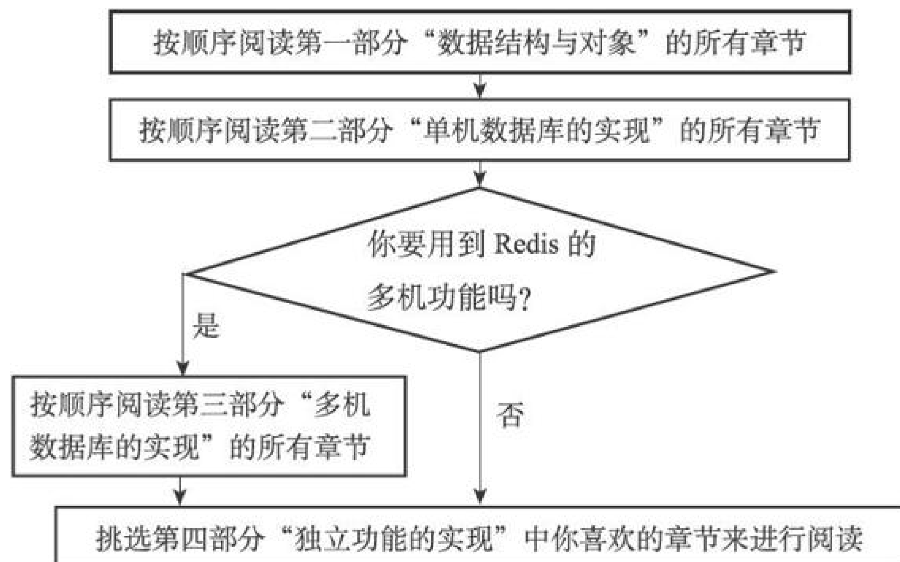


图1-1 推荐阅读方法

1.4 行文规则

名字引用规则

在第一次引用Redis源代码文件file中的名字name时，本书使用file/name格式，比如redis.c/main表示redis.c文件中的main函数，而redis.h/redisDb则表示redis.h文件中的redisDb结构，诸如此类。

另外，在第一次引用标准库头文件file中的名字name时，本书使用＜file＞/name格式，比如＜unistd.h＞/write表示unistd.h头文件的write函数，而＜stdio.h＞/printf则表示stdio.h头文件的printf函数，诸如此类。

在第一次引用某个名字之后，本书就会去掉名字前缀的文件名，直接使用名字本身。举个例子，当第一次引用redis.h文件的redisDb结构的时候，会使用redis.h/redisDb格式，而之后再次引用redisDb结构时，只使用名字redisDb。

结构引用规则

本书使用struct.property格式来引用struct结构的property属性，比如redisDb.id表示redisDb结构的id属性，而redisDb.expires则表示redisDb结构的expires属性，诸如此类。

算法规则

除非有额外说明，否则本书列出的算法复杂度一律为最坏情形下的算法复杂度。

代码规则

本书使用C语言和Python语言来展示代码：

·在描述数据结构以及比较简短的代码时，本书通常会直接粘贴Redis的源代码，也即C语言代码。

·而当需要使用代码来描述比较长或者比较复杂的程序时，本书通常会使用Python语言来表示伪代码。

本书展示的Python伪代码中通常会包含server和client两个全局变量，其中server表示服务器状态（redis.h/redisServer结构的实例），而client则表示正在执行操作的客户端状态（redis.h/redisClient结构的实例）。

1.5 配套网站

本书配套网站redisbook.com记录了本书的最新消息，并且提供了附带详细注释的Redis源代码可供下载，读者也可以通过这个网站查看和反馈本书的勘误，或者发表与本书有关的问题、意见以及建议。

第一部分　数据结构与对象

第2章　简单动态字符串

第3章　链表

第4章　字典

第5章　跳跃表

第6章　整数集合

第7章　压缩列表

第8章　对象

第2章　简单动态字符串

Redis没有直接使用C语言传统的字符串表示（以空字符结尾的字符数组，以下简称C字符串），而是自己构建了一种名为简单动态字符串（simple dynamic string，SDS）的抽象类型，并将SDS用作Redis的默认字符串表示。

在Redis里面，C字符串只会作为字符串字面量（string literal）用在一些无须对字符串值进行修改的地方，比如打印日志：

```
redisLog(REDIS_WARNING,"Redis is now ready to exit, bye bye...");
```

当Redis需要的不仅仅是一个字符串字面量，而是一个可以被修改的字符串值时，Redis就会使用SDS来表示字符串值，比如在Redis的数据库里面，包含字符串值的键值对在底层都是由SDS实现的。

举个例子，如果客户端执行命令：

```
redis> SET msg "hello world"
OK
```

那么Redis将在数据库中创建一个新的键值对，其中：

·键值对的键是一个字符串对象，对象的底层实现是一个保存着字符串“msg”的SDS。

·键值对的值也是一个字符串对象，对象的底层实现是一个保存着字符串“hello world”的SDS。

又比如，如果客户端执行命令：

```
redis> RPUSH fruits "apple" "banana" "cherry"
(integer) 3
```

那么Redis将在数据库中创建一个新的键值对，其中：

·键值对的键是一个字符串对象，对象的底层实现是一个保存了字符串“fruits”的SDS。

·键值对的值是一个列表对象，列表对象包含了三个字符串对象，这三个字符串对象分别由三个SDS实现：第一个SDS保存着字符串“apple”，第二个SDS保存着字符串“banana”，第三个SDS保存着字符串“cherry”。

除了用来保存数据库中的字符串值之外，SDS还被用作缓冲区（buffer）：AOF模块中的AOF缓冲区，以及客户端状态中的输入缓冲

区，都是由SDS实现的，在之后介绍AOF持久化和客户端状态的时候，我们会看到SDS在这两个模块中的应用。

本章接下来将对SDS的实现进行介绍，说明SDS和C字符串的不同之处，解释为什么Redis要使用SDS而不是C字符串，并在本章的最后列出SDS的操作API。

2.1 SDS的定义

每个sds.h/sdshdr结构表示一个SDS值：

```
struct sdshdr {
    //
记录buf
数组中已使用字节的数量
    //
等于SDS
所保存字符串的长度
    int len;
    //
记录buf
数组中未使用字节的数量
    int free;
    //
字节数组，用于保存字符串
    char buf[];
};
```

图2-1展示了一个SDS示例：

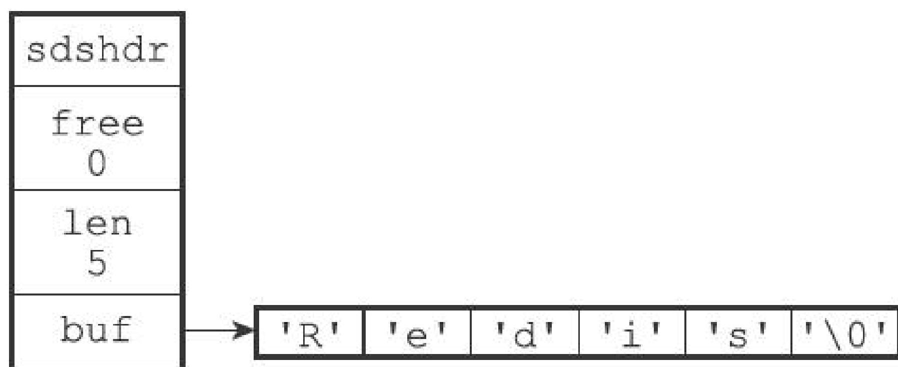


图2-1 SDS示例

·free属性的值为0，表示这个SDS没有分配任何未使用空间。

·len属性的值为5，表示这个SDS保存了一个五字节长的字符串。

·buf属性是一个char类型的数组，数组的前五个字节分别保存了'R'、'e'、'd'、'i'、's'五个字符，而最后一个字节则保存了空字符'\0'。

SDS遵循C字符串以空字符结尾的惯例，保存空字符的1字节空间不计算在SDS的len属性里面，并且为空字符分配额外的1字节空间，以及添加空字符到字符串末尾等操作，都是由SDS函数自动完成的，所以这个空字符对于SDS的使用者来说是完全透明的。遵循空字符结尾这一惯例的好处是，SDS可以直接重用一部分C字符串函数库里面的函数。

举个例子，如果我们有一个指向图2-1所示SDS的指针s，那么我们

可以直接使用＜stdio.h＞/printf函数，通过执行以下语句：

```
printf("%s", s->buf);
```

来打印出SDS保存的字符串值“Redis”，而无须为SDS编写专门的打印函数。

图2-2展示了另一个SDS示例。这个SDS和之前展示的SDS一样，都保存了字符串值“Redis”。这个SDS和之前展示的SDS的区别在于，这个SDS为buf数组分配了五字节未使用空间，所以它的free属性的值为5（图中使用五个空格来表示五字节的未使用空间）。

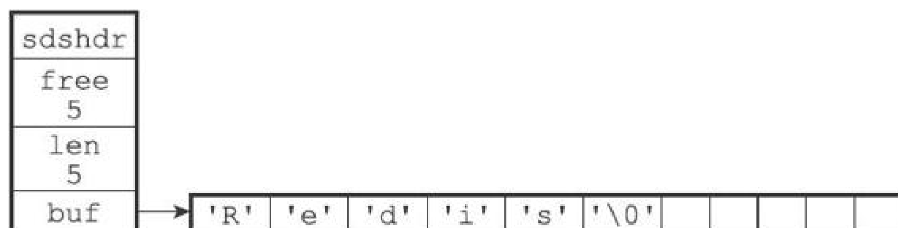


图2-2　带有未使用空间的SDS示例

接下来的一节将详细地说明未使用空间在SDS中的作用。

2.2 SDS与C字符串的区别

根据传统，C语言使用长度为N＋1的字符数组来表示长度为N的字符串，并且字符数组的最后一个元素总是空字符'\0'。

例如，图2-3就展示了一个值为"Redis"的C字符串。

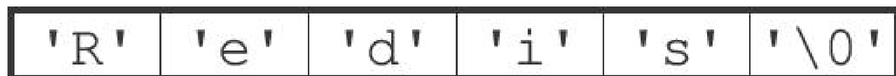


图2-3 C字符串

C语言使用的这种简单的字符串表示方式，并不能满足Redis对字符串在安全性、效率以及功能方面的要求，本节接下来的内容将详细对比C字符串和SDS之间的区别，并说明SDS比C字符串更适用于Redis的原因。

2.2.1 常数复杂度获取字符串长度

因为C字符串并不记录自身的长度信息，所以为了获取一个C字符串的长度，程序必须遍历整个字符串，对遇到的每个字符进行计数，直到遇到代表字符串结尾的空字符为止，这个操作的复杂度为O（N）。

举个例子，图2-4展示了程序计算一个C字符串长度的过程。

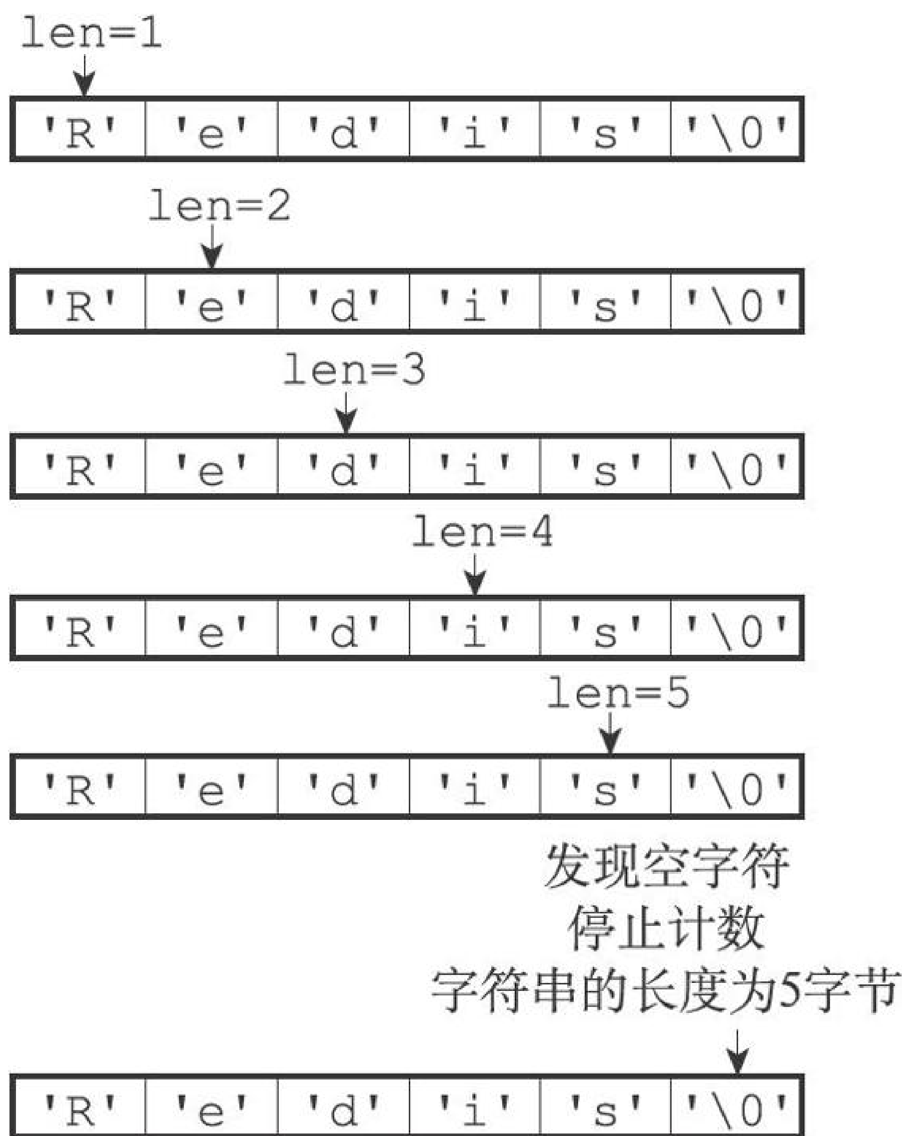


图2-4　计算C字符串长度的过程

和C字符串不同，因为SDS在len属性中记录了SDS本身的长度，所以获取一个SDS长度的复杂度仅为O（1）。

举个例子，对于图2-5所示的SDS来说，程序只要访问SDS的len属性，就可以立即知道SDS的长度为5字节。

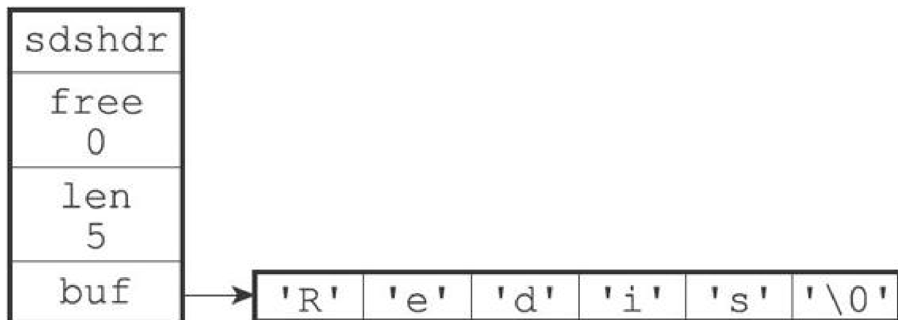


图2-5　5字节长的SDS

又例如，对于图2-6展示的SDS来说，程序只要访问SDS的len属性，就可以立即知道SDS的长度为11字节。

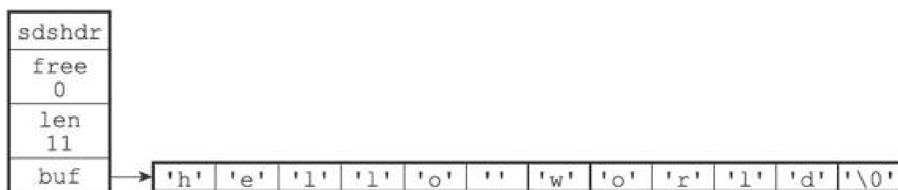


图2-6　11字节长的SDS

设置和更新SDS长度的工作是由SDS的API在执行时自动完成的，使用SDS无须进行任何手动修改长度的工作。

通过使用SDS而不是C字符串，Redis将获取字符串长度所需的复杂度从O（N）降低到了O（1），这确保了获取字符串长度的工作不会成为Redis的性能瓶颈。例如，因为字符串键在底层使用SDS来实现，所以即使我们对一个非常长的字符串键反复执行STRLEN命令，也不会对系统性能造成任何影响，因为STRLEN命令的复杂度仅为O（1）。

2.2.2　杜绝缓冲区溢出

除了获取字符串长度的复杂度高之外，C字符串不记录自身长度带来的另一个问题是容易造成缓冲区溢出（buffer　　overflow）。举个例子，<string.h>/strcat函数可以将src字符串中的内容拼接到dest字符串的末尾：

```
char *strcat(char *dest, const char *src);
```

因为C字符串不记录自身的长度，所以strcat假定用户在执行这个函数时，已经为dest分配了足够多的内存，可以容纳src字符串中的所有内容，而一旦这个假定不成立时，就会产生缓冲区溢出。

举个例子，假设程序里有两个在内存中紧邻着的C字符串s1和s2，

其中s1保存了字符串"Redis"，而s2则保存了字符串"MongoDB"，如图2-7所示。

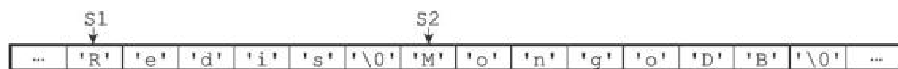


图2-7　在内存中紧邻的两个C字符串

如果一个程序员决定通过执行：

```
strcat(s1, " Cluster");
```

将s1的内容修改为"Redis　Cluster"，但粗心的他却忘了在执行strcat之前为s1分配足够的空间，那么在strcat函数执行之后，s1的数据将溢出到s2所在的空间中，导致s2保存的内容被意外地修改，如图2-8所示。



图2-8　S1的内容溢出到了S2所在的位置上

与C字符串不同，SDS的空间分配策略完全杜绝了发生缓冲区溢出的可能性：当SDS　API需要对SDS进行修改时，API会先检查SDS的空间是否满足修改所需的要求，如果不满足的话，API会自动将SDS的空间扩展至执行修改所需的大小，然后才执行实际的修改操作，所以使用SDS既不需要手动修改SDS的空间大小，也不会出现前面所说的缓冲区溢出问题。

举个例子，SDS的API里面也有一个用于执行拼接操作的sdscat函数，它可以将一个C字符串拼接到给定SDS所保存的字符串的后面，但是在执行拼接操作之前，sdscat会先检查给定SDS的空间是否足够，如果不够的话，sdscat就会先扩展SDS的空间，然后才执行拼接操作。

例如，如果我们执行：

```
sdscat(s, " Cluster");
```

其中SDS值s如图2-9所示，那么sdscat将在执行拼接操作之前检查s的长度是否足够，在发现s目前的空间不足以拼接"Cluster"之后，sdscat就会先扩展s的空间，然后才执行拼接"Cluster"的操作，拼接操作完成之后的SDS如图2-10所示。

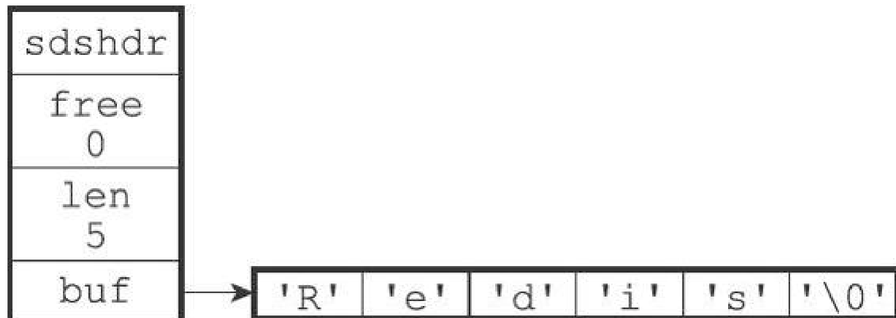


图2-9 sdscat执行之前的SDS

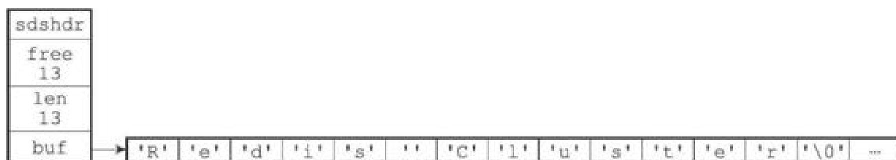


图2-10　sdscat执行之后的SDS

注意，图2-10所示的SDS，sdscat不仅对这个SDS进行了拼接操作，它还为SDS分配了13字节的未使用空间，并且拼接之后的字符串也正好是13字节长，这种现象既不是bug也不是巧合，它和SDS的空间分配策略有关，接下来的小节将对这一策略进行说明。

2.2.3　减少修改字符串时带来的内存重分配次数

正如前两个小节所说，因为C字符串并不记录自身的长度，所以对于一个包含了N个字符的C字符串来说，这个C字符串的底层实现总是一个N+1个字符长的数组（额外的一个字符空间用于保存空字符）。因为C字符串的长度和底层数组的长度之间存在着这种关联性，所以每次增长或者缩短一个C字符串，程序都总要对保存这个C字符串的数组进行一次内存重分配操作：

·如果程序执行的是增长字符串的操作，比如拼接操作（append），那么在执行这个操作之前，程序需要先通过内存重分配来扩展底层数组的空间大小——如果忘了这一步就会产生缓冲区溢出。

·如果程序执行的是缩短字符串的操作，比如截断操作（trim），那么在执行这个操作之后，程序需要通过内存重分配来释放字符串不再使用的那部分空间——如果忘了这一步就会产生内存泄漏。

举个例子，如果我们持有一个值为"Redis"的C字符串s，那么为了将s的值改为"Redis Cluster"，在执行：

```
strcat(s, " Cluster");
```

之前，我们需要先使用内存重分配操作，扩展s的空间。

之后，如果我们又打算将s的值从"Redis Cluster"改为"Redis Cluster Tutorial"，那么在执行：

```
strcat(s, " Tutorial");
```

之前，我们需要再次使用内存重分配扩展s的空间，诸如此类。

因为内存重分配涉及复杂的算法，并且可能需要执行系统调用，所以它通常是一个比较耗时的操作：

·在一般程序中，如果修改字符串长度的情况不太常出现，那么每次修改都执行一次内存重分配是可以接受的。

·但是Redis作为数据库，经常被用于速度要求严苛、数据被频繁修改的场合，如果每次修改字符串的长度都需要执行一次内存重分配的话，那么光是执行内存重分配的时间就会占去修改字符串所用时间的一大部分，如果这种修改频繁地发生的话，可能还会对性能造成影响。

为了避免C字符串的这种缺陷，SDS通过未使用空间解除了字符串长度和底层数组长度之间的关联：在SDS中，buf数组的长度不一定就是字符数量加一，数组里面可以包含未使用的字节，而这些字节的数量就由SDS的free属性记录。

通过未使用空间，SDS实现了空间预分配和惰性空间释放两种优化策略。

1.空间预分配

空间预分配用于优化SDS的字符串增长操作：当SDS的API对一个SDS进行修改，并且需要对SDS进行空间扩展的时候，程序不仅会为SDS分配修改所必须要的空间，还会为SDS分配额外的未使用空间。

其中，额外分配的未使用空间数量由以下公式决定：

·如果对SDS进行修改之后，SDS的长度（也即是len属性的值）将小于1MB，那么程序分配和len属性同样大小的未使用空间，这时SDS len属性的值将和free属性的值相同。举个例子，如果进行修改之后，SDS的len将变成13字节，那么程序也会分配13字节的未使用空间，SDS的buf数组的实际长度将变成13＋13＋1＝27字节（额外的一字节用于保存空字符）。

·如果对SDS进行修改之后，SDS的长度将大于等于1MB，那么程序会分配1MB的未使用空间。举个例子，如果进行修改之后，SDS的len将变成30MB，那么程序会分配1MB的未使用空间，SDS的buf数组的实际长度将为30MB＋1MB＋1byte。

通过空间预分配策略，Redis可以减少连续执行字符串增长操作所需的内存重分配次数。

举个例子，对于图2-11所示的SDS值s来说，如果我们执行：

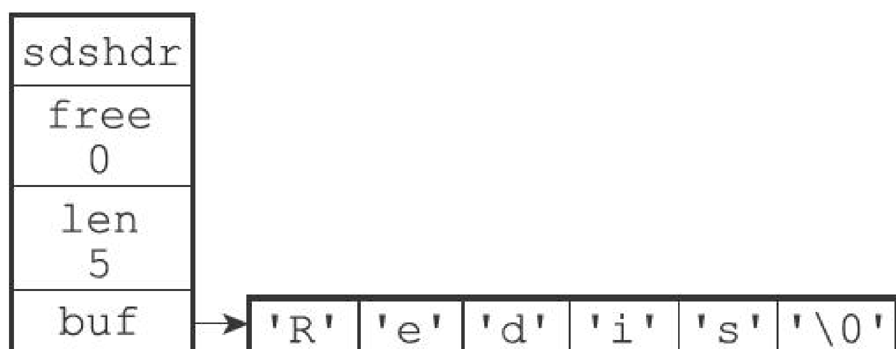


图2-11　执行sdscat之前的SDS

```
sdscat(s, " Cluster");
```

那么sdscat将执行一次内存重分配操作，将SDS的长度修改为13字节，并将SDS的未使用空间同样修改为13字节，如图2-12所示。

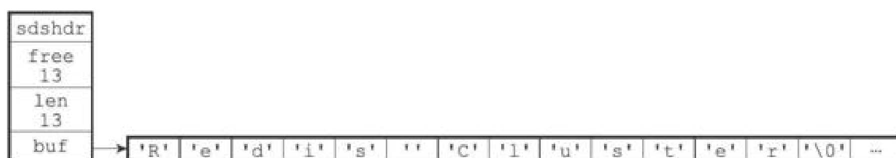


图2-12　执行sdscat之后SDS

如果这时，我们再次对s执行：

```
sdscat(s, " Tutorial");
```

那么这次sdscat将不需要执行内存重分配，因为未使用空间里面的13字节足以保存9字节的"Tutorial"，执行sdscat之后的SDS如图2-13所示。

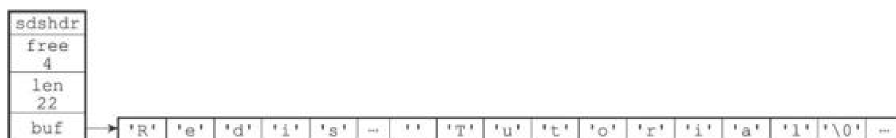


图2-13　再次执行sdscat之后的SDS

在扩展SDS空间之前，SDS API会先检查未使用空间是否足够，如果足够的话，API就会直接使用未使用空间，而无须执行内存重分配。

通过这种预分配策略，SDS将连续增长N次字符串所需的内存重分配次数从必定N次降低为最多N次。

2.惰性空间释放

惰性空间释放用于优化SDS的字符串缩短操作：当SDS的API需要缩短SDS保存的字符串时，程序并不立即使用内存重分配来回收缩短后多出来的字节，而是使用free属性将这些字节的数量记录起来，并等待将来使用。

举个例子，sdstrim函数接受一个SDS和一个C字符串作为参数，移除SDS中所有在C字符串中出现过的字符。

比如对于图2-14所示的SDS值s来说，执行：

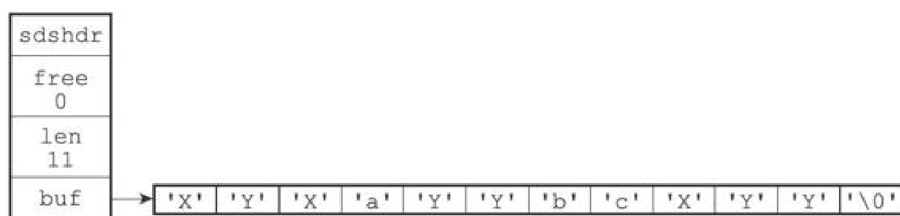


图2-14　执行sdstrim之前的SDS

```
sdstrim(s, "XY"); //
移除SDS
字符串中的所有'X'
和'Y'
```

会将SDS修改成图2-15所示的样子。

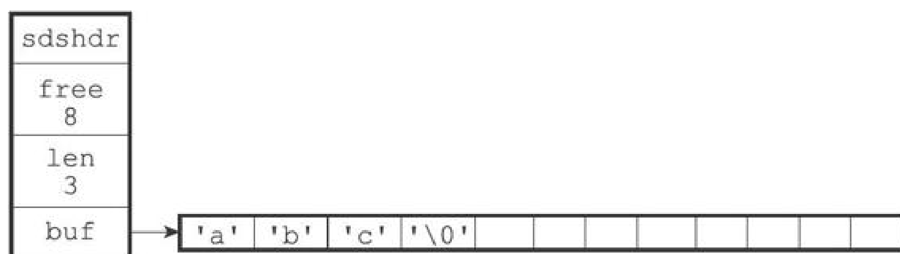


图2-15　执行sdstrim之后的SDS

注意执行sdstrim之后的SDS并没有释放多出来的8字节空间，而是将这8字节空间作为未使用空间保留在了SDS里面，如果将来要对SDS进行增长操作的话，这些未使用空间就可能会派上用场。

举个例子，如果现在对s执行：

```
sdscat(s, " Redis");
```

那么完成这次sdscat操作将不需要执行内存重分配：因为SDS里面预留的8字节空间已经足以拼接6个字节长的"Redis"，如图2-16所示。

通过惰性空间释放策略，SDS避免了缩短字符串时所需的内存重分配操作，并为将来可能有的增长操作提供了优化。

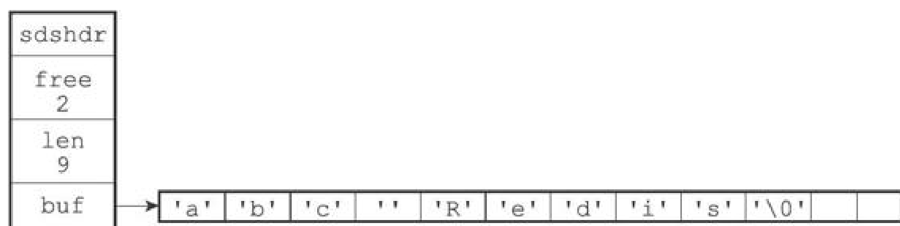


图2-16　执行sdscat之后的SDS

与此同时，SDS也提供了相应的API，让我们可以在有需要时，真正地释放SDS的未使用空间，所以不用担心惰性空间释放策略会造成内存浪费。

2.2.4　二进制安全

C字符串中的字符必须符合某种编码（比如ASCII），并且除了字符串的末尾之外，字符串里面不能包含空字符，否则最先被程序读入的空字符将被误认为是字符串结尾，这些限制使得C字符串只能保存文本数据，而不能保存像图片、音频、视频、压缩文件这样的二进制数据。

举个例子，如果有一种使用空字符来分割多个单词的特殊数据格式，如图2-17所示，那么这种格式就不能使用C字符串来保存，因为C字符串所用的函数只会识别出其中的"Redis"，而忽略之后的"Cluster"。



图2-17　使用空字符来分割单词的特殊数据格式

虽然数据库一般用于保存文本数据，但使用数据库来保存二进制数据的场景也不少见，因此，为了确保Redis可以适用于各种不同的使用场景，SDS的API都是二进制安全的（binary-safe），所有SDS　API都会以处理二进制的方式来处理SDS存放在buf数组里的数据，程序不会对其中的数据做任何限制、过滤、或者假设，数据在写入时是什么样的，它被读取时就是什么样。

这也是我们将SDS的buf属性称为字节数组的原因——Redis不是用这个数组来保存字符，而是用它来保存一系列二进制数据。

例如，使用SDS来保存之前提到的特殊数据格式就没有任何问题，因为SDS使用len属性的值而不是空字符来判断字符串是否结束，如图2-18所示。

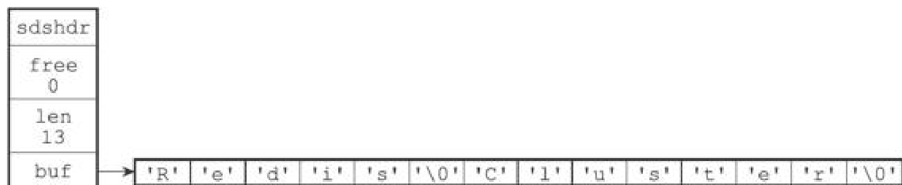


图2-18　保存了特殊数据格式的SDS

通过使用二进制安全的SDS，而不是C字符串，使得Redis不仅可以保存文本数据，还可以保存任意格式的二进制数据。

2.2.5　兼容部分C字符串函数

虽然SDS的API都是二进制安全的，但它们一样遵循C字符串以空字符结尾的惯例：这些API总会将SDS保存的数据的末尾设置为空字符，并且总会在为buf数组分配空间时多分配一个字节来容纳这个空字符，这是为了让那些保存文本数据的SDS可以重用一部分＜string.h＞库定义的函数。

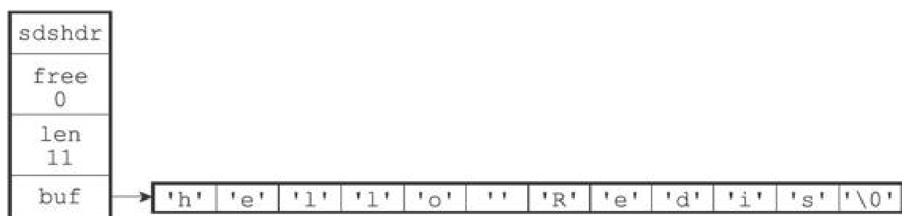


图2-19　一个保存着文本数据的SDS

举个例子，如图2-19所示，如果我们有一个保存文本数据的SDS值sds，那么我们就可以重用＜string.h＞/strcasecmp函数，使用它来对比SDS保存的字符串和另一个C字符串：

```
strcasecmp(sds->buf, "hello world");
```

这样Redis就不用自己专门去写一个函数来对比SDS值和C字符串值了。

与此类似，我们还可以将一个保存文本数据的SDS作为strcat函数的第二个参数，将SDS保存的字符串追加到一个C字符串的后面：

```
strcat(c_string, sds->buf);
```

这样Redis就不用专门编写一个将SDS字符串追加到C字符串之后的函数了。

通过遵循C字符串以空字符结尾的惯例，SDS可以在有需要时重用

<string.h>函数库，从而避免了不必要的代码重复。

2.2.6 总结

表2-1对C字符串和SDS之间的区别进行了总结。

表2-1 C字符串和SDS之间的区别

C 字符串	SDS
获取字符串长度的复杂度为 $O(N)$	获取字符串长度的复杂度为 $O(1)$
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会造成缓冲区溢出
修改字符串长度 N 次必然需要执行 N 次内存重分配	修改字符串长度 N 次最多需要执行 N 次内存重分配
只能保存文本数据	可以保存文本或者二进制数据
可以使用所有 <string.h> 库中的函数	可以使用一部分 <string.h> 库中的函数

2.3 SDS API

表2-2列出了SDS的主要操作API。

表2-2 SDS的主要操作API

函 数	作 用	时间复杂度
sdsnew	创建一个包含给定 C 字符串的 SDS	$O(N)$，N 为给定 C 字符串的长度
sdsempty	创建一个不包含任何内容的空 SDS	$O(1)$
sdsfree	释放给定的 SDS	$O(N)$，N 为被释放 SDS 的长度
sdslen	返回 SDS 的已使用空间字节数	这个值可以通过读取 SDS 的 `len` 属性来直接获得，复杂度为 $O(1)$
sdsavail	返回 SDS 的未使用空间字节数	这个值可以通过读取 SDS 的 `free` 属性来直接获得，复杂度为 $O(1)$
sdsdup	创建一个给定 SDS 的副本（copy）	$O(N)$，N 为给定 SDS 的长度
sdsclear	清空 SDS 保存的字符串内容	因为惰性空间释放策略，复杂度为 $O(1)$
sdscat	将给定 C 字符串拼接到 SDS 字符串的末尾	$O(N)$，N 为被拼接 C 字符串的长度
sdscatsds	将给定 SDS 字符串拼接到另一个 SDS 字符串的末尾	$O(N)$，N 为被拼接 SDS 字符串的长度
sdscpy	将给定的 C 字符串复制到 SDS 里面，覆盖 SDS 原有的字符串	$O(N)$，N 为被复制 C 字符串的长度
sdsgrowzero	用空字符将 SDS 扩展至给定长度	$O(N)$，N 为扩展新增的字节数
sdsrange	保留 SDS 给定区间内的数据，不在区间内的数据会被覆盖或清除	$O(N)$，N 为被保留数据的字节数
sdstrim	接受一个 SDS 和一个 C 字符串作为参数，从 SDS 中移除所有在 C 字符串中出现过的字符	$O(N^2)$，N 为给定 C 字符串的长度
sdscmp	对比两个 SDS 字符串是否相同	$O(N)$，N 为两个 SDS 中较短的那个 SDS 的长度

2.4 重点回顾

·Redis只会使用C字符串作为字面量，在大多数情况下，Redis使用SDS（Simple Dynamic String，简单动态字符串）作为字符串表示。

·比起C字符串，SDS具有以下优点：

1）常数复杂度获取字符串长度。

2）杜绝缓冲区溢出。

3）减少修改字符串长度时所需的内存重分配次数。

4）二进制安全。

5）兼容部分C字符串函数。

2.5 参考资料

·《C语言接口与实现：创建可重用软件的技术》一书的第15章和第16章介绍了一个和SDS类似的通用字符串实现。

·维基百科的Binary Safe词条（http://en.wikipedia.org/wiki/Binary-safe）和http://computer.yourdictionary.com/binary-safe给出了二进制安全的定义。

·维基百科的Null-terminated string词条给出了空字符结尾字符串的定义，说明了这种表示的来源，以及C语言使用这种字符串表示的历史原因：http://en.wikipedia.org/wiki/Null-terminated_string

·《C标准库》一书的第14章给出了标准库所有API的介绍，以及这些API的基础实现。

·GNU C库的主页上提供了GNU C标准库的下载包，其中的/string文件夹包含了所有API的完整实现：http://www.gnu.org/software/libc

第3章　链表

链表提供了高效的节点重排能力，以及顺序性的节点访问方式，并且可以通过增删节点来灵活地调整链表的长度。

作为一种常用数据结构，链表内置在很多高级的编程语言里面，因为Redis使用的C语言并没有内置这种数据结构，所以Redis构建了自己的链表实现。

链表在Redis中的应用非常广泛，比如列表键的底层实现之一就是链表。当一个列表键包含了数量比较多的元素，又或者列表中包含的元素都是比较长的字符串时，Redis就会使用链表作为列表键的底层实现。

举个例子，以下展示的integers列表键包含了从1到1024共一千零二十四个整数：

```
redis> LLEN integers
(integer) 1024
redis> LRANGE integers 0 10
1)"1"
2)"2"
3)"3"
4)"4"
5)"5"
6)"6"
7)"7"
8)"8"
9)"9"
10)"10"
11)"11"
```

integers列表键的底层实现就是一个链表，链表中的每个节点都保存了一个整数值。

除了链表键之外，发布与订阅、慢查询、监视器等功能也用到了链表，Redis服务器本身还使用链表来保存多个客户端的状态信息，以及使用链表来构建客户端输出缓冲区（output　buffer），本书后续的章节将陆续对这些链表应用进行介绍。

本章接下来的内容将对Redis的链表实现进行介绍，并列出相应的链表和链表节点API。

因为已经有很多优秀的算法书籍对链表的基本定义和相关算法进行了详细的讲解，所以本章不会介绍这些内容，如果不具备关于链表的基本知识的话，可以参考《算法：C语言实现（第1～4部分）》一书的3.3至3.5节，或者《数据结构与算法分析：C语言描述》一书的3.2节，又或者《算法导论（第三版）》一书的10.2节。

3.1 链表和链表节点的实现

每个链表节点使用一个adlist.h/listNode结构来表示：

```
typedef struct listNode {
    //
前置节点
    struct listNode * prev;
    //
后置节点
    struct listNode * next;
    //
节点的值
    void * value;
}listNode;
```

多个listNode可以通过prev和next指针组成双端链表，如图3-1所示。

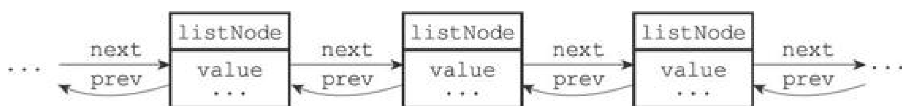


图3-1　由多个listNode组成的双端链表

虽然仅仅使用多个listNode结构就可以组成链表，但使用adlist.h/list来持有链表的话，操作起来会更方便：

```
typedef struct list {
    //
表头节点
    listNode * head;
    //
表尾节点
    listNode * tail;
    //
链表所包含的节点数量
    unsigned long len;
    //
节点值复制函数
    void *(*dup)(void *ptr);
    //
节点值释放函数
    void (*free)(void *ptr);
    //
节点值对比函数
    int (*match)(void *ptr,void *key);
} list;
```

list结构为链表提供了表头指针head、表尾指针tail，以及链表长度计数器len，而dup、free和match成员则是用于实现多态链表所需的类型特定函数：

·dup函数用于复制链表节点所保存的值；

·free函数用于释放链表节点所保存的值；

·match函数则用于对比链表节点所保存的值和另一个输入值是否相等。

图3-2是由一个list结构和三个listNode结构组成的链表。

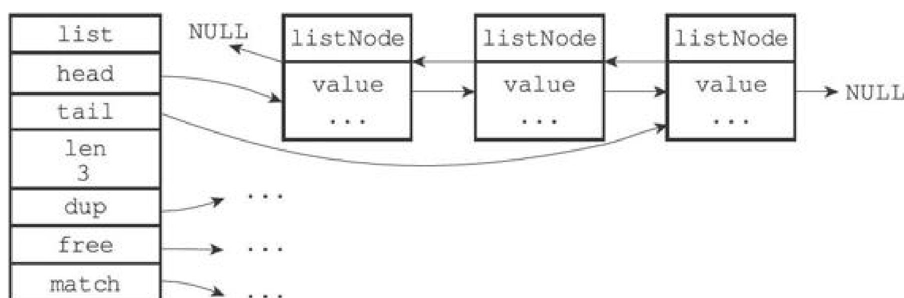


图3-2　由list结构和listNode结构组成的链表

Redis的链表实现的特性可以总结如下：

·双端：链表节点带有prev和next指针，获取某个节点的前置节点和后置节点的复杂度都是O（1）。

·无环：表头节点的prev指针和表尾节点的next指针都指向NULL，对链表的访问以NULL为终点。

·带表头指针和表尾指针：通过list结构的head指针和tail指针，程序获取链表的表头节点和表尾节点的复杂度为O（1）。

·带链表长度计数器：程序使用list结构的len属性来对list持有的链表节点进行计数，程序获取链表中节点数量的复杂度为O（1）。

·多态：链表节点使用void*指针来保存节点值，并且可以通过list结构的dup、free、match三个属性为节点值设置类型特定函数，所以链表可以用于保存各种不同类型的值。

3.2　链表和链表节点的API

表3-1列出了所有用于操作链表和链表节点的API。

表3-1　链表和链表节点API

函数	作用	时间复杂度
listSetDupMethod	将给定的函数设置为链表的节点值复制函数	复制函数可以通过链表的 dup 属性直接获得，$O(1)$
listGetDupMethod	返回链表当前正在使用的节点值复制函数	$O(1)$
listSetFreeMethod	将给定的函数设置为链表的节点值释放函数	释放函数可以通过链表的 free 属性直接获得，$O(1)$
listGetFree	返回链表当前正在使用的节点值释放函数	$O(1)$
listSetMatchMethod	将给定的函数设置为链表的节点值对比函数	对比函数可以通过链表的 match 属性直接获得，$O(1)$
listGetMatchMethod	返回链表当前正在使用的节点值对比函数	$O(1)$
listLength	返回链表的长度（包含了多少个节点）	链表长度可以通过链表的 len 属性直接获得，$O(1)$
listFirst	返回链表的表头节点	表头节点可以通过链表的 head 属性直接获得，$O(1)$
listLast	返回链表的表尾节点	表尾节点可以通过链表的 tail 属性直接获得，$O(1)$
listPrevNode	返回给定节点的前置节点	前置节点可以通过节点的 prev 属性直接获得，$O(1)$
listNextNode	返回给定节点的后置节点	后置节点可以通过节点的 next 属性直接获得，$O(1)$
listNodeValue	返回给定节点目前正在保存的值	节点值可以通过节点的 value 属性直接获得，$O(1)$
listCreate	创建一个不包含任何节点的新链表	$O(1)$
listAddNodeHead	将一个包含给定值的新节点添加到给定链表的表头	$O(1)$
listAddNodeTail	将一个包含给定值的新节点添加到给定链表的表尾	$O(1)$
listInsertNode	将一个包含给定值的新节点添加到给定节点的之前或者之后	$O(1)$
listSearchKey	查找并返回链表中包含给定值的节点	$O(N)$，N 为链表长度
listIndex	返回链表在给定索引上的节点	$O(N)$，N 为链表长度
listDelNode	从链表中删除给定节点	$O(N)$，N 为链表长度
listRotate	将链表的表尾节点弹出，然后将被弹出的节点插入到链表的表头，成为新的表头节点	$O(1)$
listDup	复制一个给定链表的副本	$O(N)$，N 为链表长度
listRelease	释放给定链表，以及链表中的所有节点	$O(N)$，N 为链表长度

3.3 重点回顾

·链表被广泛用于实现Redis的各种功能，比如列表键、发布与订阅、慢查询、监视器等。

·每个链表节点由一个listNode结构来表示，每个节点都有一个指向前置节点和后置节点的指针，所以Redis的链表实现是双端链表。

·每个链表使用一个list结构来表示，这个结构带有表头节点指针、表尾节点指针，以及链表长度等信息。

·因为链表表头节点的前置节点和表尾节点的后置节点都指向NULL，所以Redis的链表实现是无环链表。

·通过为链表设置不同的类型特定函数，Redis的链表可以用于保存各种不同类型的值。

第4章　字典

字典，又称为符号表（symbol table）、关联数组（associative array）或映射（map），是一种用于保存键值对（key-value pair）的抽象数据结构。

在字典中，一个键（key）可以和一个值（value）进行关联（或者说将键映射为值），这些关联的键和值就称为键值对。

字典中的每个键都是独一无二的，程序可以在字典中根据键查找与之关联的值，或者通过键来更新值，又或者根据键来删除整个键值对，等等。

字典经常作为一种数据结构内置在很多高级编程语言里面，但Redis所使用的C语言并没有内置这种数据结构，因此Redis构建了自己的字典实现。

字典在Redis中的应用相当广泛，比如Redis的数据库就是使用字典来作为底层实现的，对数据库的增、删、查、改操作也是构建在对字典的操作之上的。

举个例子，当我们执行命令：

```
redis> SET msg "hello world"
OK
```

在数据库中创建一个键为"msg"，值为"hello world"的键值对时，这个键值对就是保存在代表数据库的字典里面的。

除了用来表示数据库之外，字典还是哈希键的底层实现之一，当一个哈希键包含的键值对比较多，又或者键值对中的元素都是比较长的字符串时，Redis就会使用字典作为哈希键的底层实现。

举个例子，website是一个包含10086个键值对的哈希键，这个哈希键的键都是一些数据库的名字，而键的值就是数据库的主页网址：

```
redis> HLEN website
(integer) 10086
redis> HGETALL website
1)"Redis"
2)"Redis.io"
3)"MariaDB"
4)"MariaDB.org"
5)"MongoDB"
6)"MongoDB.org"
# ...
```

website键的底层实现就是一个字典，字典中包含了10086个键值

对，例如：

·键值对的键为"Redis"，值为"Redis.io"。

·键值对的键为"MariaDB"，值为"MariaDB.org"；

·键值对的键为"MongoDB"，值为"MongoDB.org"；

除了用来实现数据库和哈希键之外，Redis的不少功能也用到了字典，在后续的章节中会不断地看到字典在Redis中的各种不同应用。

本章接下来的内容将对Redis的字典实现进行详细介绍，并列出字典的操作API。本章不会对字典的基本定义和基础算法进行介绍，如果有需要的话，可以参考以下这些资料：

·维基百科的Associative Array词条（http://en.wikipedia.org/wiki/Associative_array）和Hash Table词条（http://en.wikipedia.org/wiki/Hash_table）。

·《算法：C语言实现（第1～4部分）》一书的第14章。

·《算法导论（第三版）》一书的第11章。

4.1 字典的实现

Redis的字典使用哈希表作为底层实现，一个哈希表里面可以有多个哈希表节点，而每个哈希表节点就保存了字典中的一个键值对。

接下来的三个小节将分别介绍Redis的哈希表、哈希表节点以及字典的实现。

4.1.1 哈希表

Redis字典所使用的哈希表由dict.h/dictht结构定义：

```
typedef struct dictht {
    //
哈希表数组
    dictEntry **table;
    //
哈希表大小
    unsigned long size;
    //
哈希表大小掩码，用于计算索引值
    //
总是等于size-1
    unsigned long sizemask;
    //
该哈希表已有节点的数量
    unsigned long used;
} dictht;
```

table属性是一个数组，数组中的每个元素都是一个指向dict.h/dictEntry结构的指针，每个dictEntry结构保存着一个键值对。size属性记录了哈希表的大小，也即是table数组的大小，而used属性则记录了哈希表目前已有节点（键值对）的数量。sizemask属性的值总是等于size-1，这个属性和哈希值一起决定一个键应该被放到table数组的哪个索引上面。

图4-1展示了一个大小为4的空哈希表（没有包含任何键值对）。

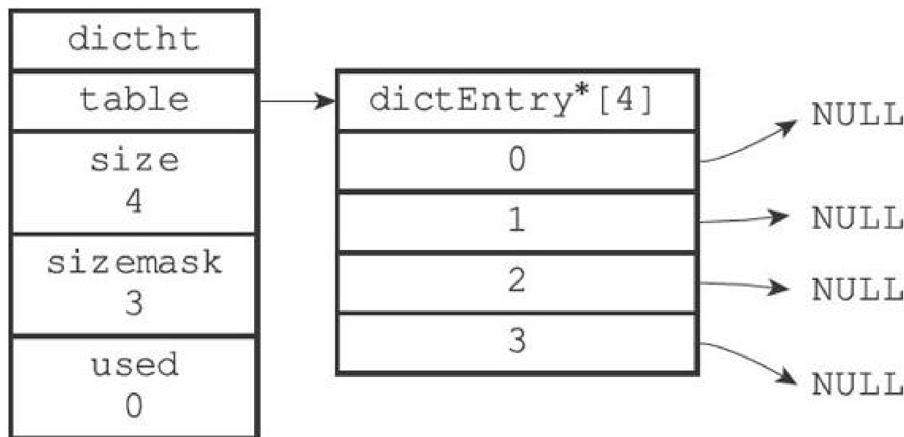


图4-1　一个空的哈希表

4.1.2　哈希表节点

哈希表节点使用dictEntry结构表示，每个dictEntry结构都保存着一个键值对：

```
typedef struct dictEntry {
    //
键
    void *key;
    //
值
    union{
        void *val;
        uint64_tu64;
        int64_ts64;
    } v;
    //
指向下个哈希表节点，形成链表
    struct dictEntry *next;
} dictEntry;
```

key属性保存着键值对中的键，而v属性则保存着键值对中的值，其中键值对的值可以是一个指针，或者是一个uint64_t整数，又或者是一个int64_t整数。

next属性是指向另一个哈希表节点的指针，这个指针可以将多个哈希值相同的键值对连接在一次，以此来解决键冲突（collision）的问题。

举个例子，图4-2就展示了如何通过next指针，将两个索引值相同的键k1和k0连接在一起。

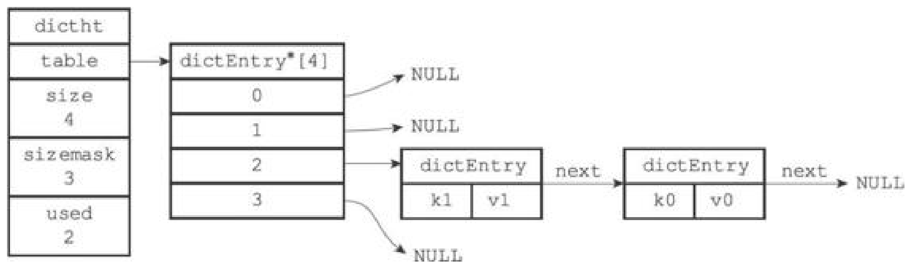


图4-2　连接在一起的键K1和键K0

4.1.3　字典

Redis中的字典由dict.h/dict结构表示：

```
typedef struct dict {
    //
类型特定函数
    dictType *type;
    //
私有数据
    void *privdata;
    //
哈希表
    dictht ht[2];
    // rehash
索引
    //
当rehash
不在进行时，值为-1
    in trehashidx; /* rehashing not in progress if rehashidx == -1 */
} dict;
```

type属性和privdata属性是针对不同类型的键值对，为创建多态字典而设置的：

·type属性是一个指向dictType结构的指针，每个dictType结构保存了一簇用于操作特定类型键值对的函数，Redis会为用途不同的字典设置不同的类型特定函数。

·而privdata属性则保存了需要传给那些类型特定函数的可选参数。

```
typedef struct dictType {
    //
计算哈希值的函数
    unsigned int (*hashFunction)(const void *key);
    //
复制键的函数
    void *(*keyDup)(void *privdata, const void *key);
    //
复制值的函数
    void *(*valDup)(void *privdata, const void *obj);
    //
对比键的函数
    int (*keyCompare)(void *privdata, const void *key1, const void *key2);
    //
销毁键的函数
    void (*keyDestructor)(void *privdata, void *key);
```

```
    //
销毁值的函数
    void (*valDestructor)(void *privdata, void *obj);
} dictType;
```

ht属性是一个包含两个项的数组，数组中的每个项都是一个dictht哈希表，一般情况下，字典只使用ht[0]哈希表，ht[1]哈希表只会在对ht[0]哈希表进行rehash时使用。

除了ht[1]之外，另一个和rehash有关的属性就是rehashidx，它记录了rehash目前的进度，如果目前没有在进行rehash，那么它的值为-1。

图4-3展示了一个普通状态下（没有进行rehash）的字典。

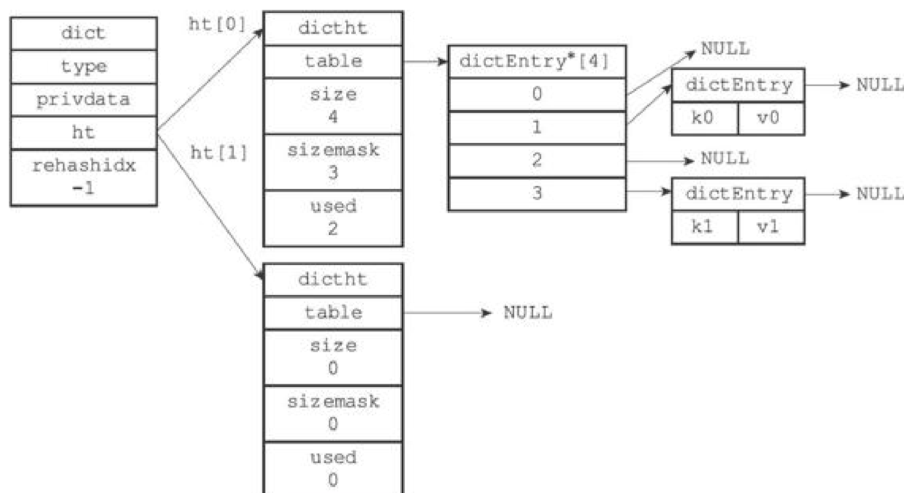


图4-3　普通状态下的字典

4.2 哈希算法

当要将一个新的键值对添加到字典里面时，程序需要先根据键值对的键计算出哈希值和索引值，然后再根据索引值，将包含新键值对的哈希表节点放到哈希表数组的指定索引上面。

Redis计算哈希值和索引值的方法如下：

```
#
使用字典设置的哈希函数，计算键key
的哈希值
hash = dict->type->hashFunction(key);
#
使用哈希表的sizemask
属性和哈希值，计算出索引值
#
根据情况不同，ht[x]
可以是ht[0]
或者ht[1]
index = hash & dict->ht[x].sizemask;
```

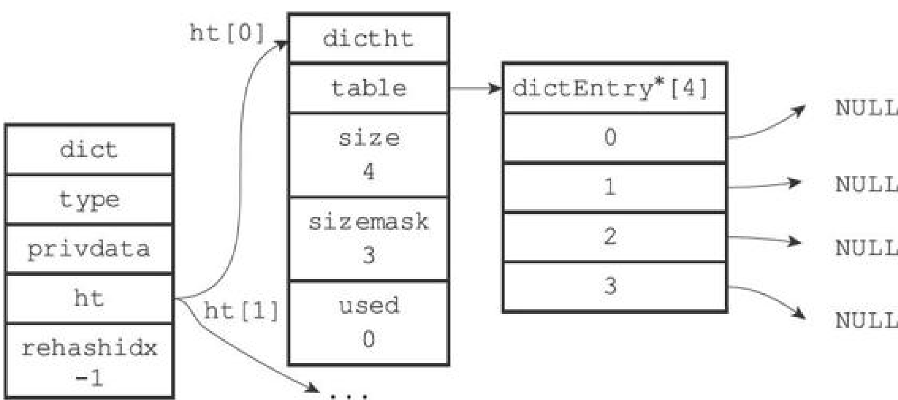


图4-4　空字典

举个例子，对于图4-4所示的字典来说，如果我们要将一个键值对k0和v0添加到字典里面，那么程序会先使用语句：

```
hash = dict->type->hashFunction(k0);
```

计算键k0的哈希值。

假设计算得出的哈希值为8，那么程序会继续使用语句：

```
index = hash&dict->ht[0].sizemask = 8 & 3 = 0;
```

计算出键k0的索引值0，这表示包含键值对k0和v0的节点应该被放置到哈希表数组的索引0位置上，如图4-5所示。

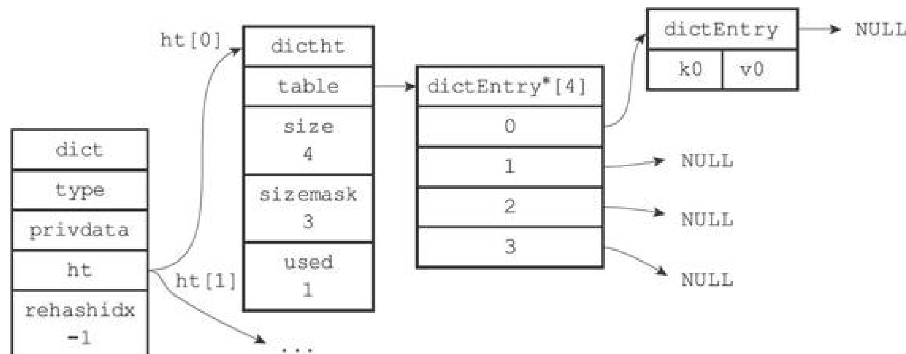


图4-5　添加键值对K0和v0之后的字典

当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。

MurmurHash算法最初由Austin　Appleby于2008年发明，这种算法的优点在于，即使输入的键是有规律的，算法仍能给出一个很好的随机分布性，并且算法的计算速度也非常快。

MurmurHash算法目前的最新版本为MurmurHash3，而Redis使用的是MurmurHash2，关于MurmurHash算法的更多信息可以参考该算法的主页：http://code.google.com/p/smhasher/。

4.3 解决键冲突

当有两个或以上数量的键被分配到了哈希表数组的同一个索引上面时，我们称这些键发生了冲突（collision）。

Redis的哈希表使用链地址法（separate chaining）来解决键冲突，每个哈希表节点都有一个next指针，多个哈希表节点可以用next指针构成一个单向链表，被分配到同一个索引上的多个节点可以用这个单向链表连接起来，这就解决了键冲突的问题。

举个例子，假设程序要将键值对k2和v2添加到图4-6所示的哈希表里面，并且计算得出k2的索引值为2，那么键k1和k2将产生冲突，而解决冲突的办法就是使用next指针将键k2和k1所在的节点连接起来，如图4-7所示。

因为dictEntry节点组成的链表没有指向链表表尾的指针，所以为了速度考虑，程序总是将新节点添加到链表的表头位置（复杂度为O（1）），排在其他已有节点的前面。

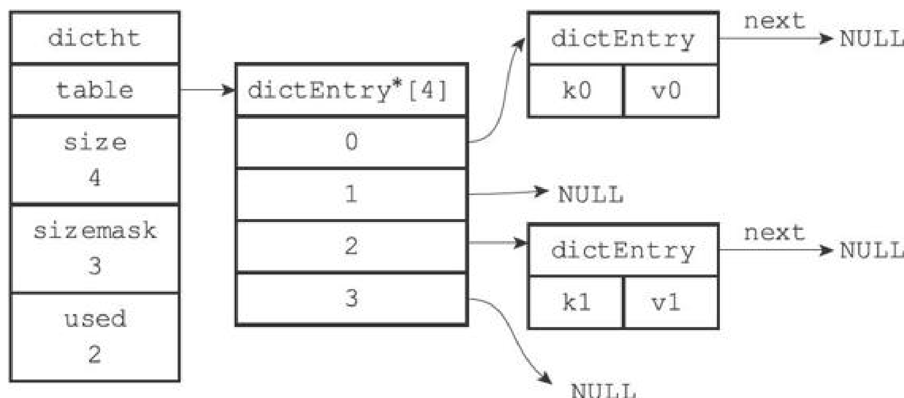


图4-6　一个包含两个键值对的哈希表

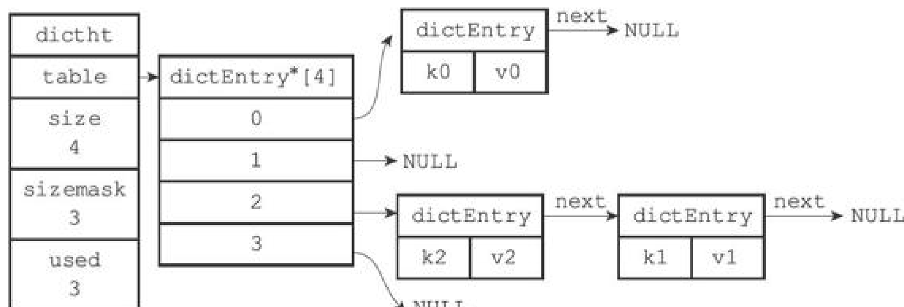


图4-7　使用链表解决k2和k1的冲突

4.4 rehash

随着操作的不断执行，哈希表保存的键值对会逐渐地增多或者减少，为了让哈希表的负载因子（load factor）维持在一个合理的范围之内，当哈希表保存的键值对数量太多或者太少时，程序需要对哈希表的大小进行相应的扩展或者收缩。

扩展和收缩哈希表的工作可以通过执行rehash（重新散列）操作来完成，Redis对字典的哈希表执行rehash的步骤如下：

1）为字典的ht[1]哈希表分配空间，这个哈希表的空间大小取决于要执行的操作，以及ht[0]当前包含的键值对数量（也即是ht[0].used属性的值）：

·如果执行的是扩展操作，那么ht[1]的大小为第一个大于等于ht[0].used*2的2^n（2的n次方幂）；

·如果执行的是收缩操作，那么ht[1]的大小为第一个大于等于ht[0].used的2^n。

2）将保存在ht[0]中的所有键值对rehash到ht[1]上面：rehash指的是重新计算键的哈希值和索引值，然后将键值对放置到ht[1]哈希表的指定位置上。

3）当ht[0]包含的所有键值对都迁移到了ht[1]之后（ht[0]变为空表），释放ht[0]，将ht[1]设置为ht[0]，并在ht[1]新创建一个空白哈希表，为下一次rehash做准备。

举个例子，假设程序要对图4-8所示字典的ht[0]进行扩展操作，那么程序将执行以下步骤：

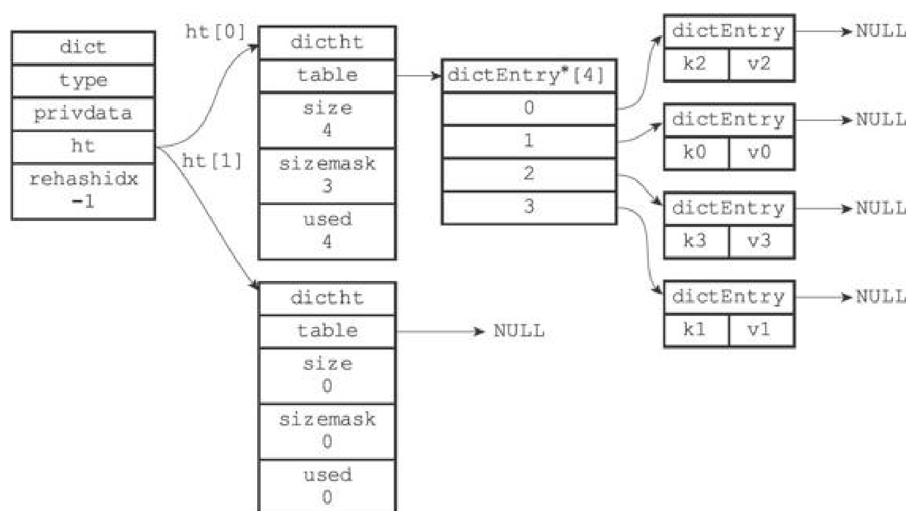


图4-8　执行rehash之前的字典

1）ht[0].used当前的值为4，4*2=8，而8（2^3）恰好是第一个大于等于4的2的n次方，所以程序会将ht[1]哈希表的大小设置为8。图4-9展示了ht[1]在分配空间之后，字典的样子。

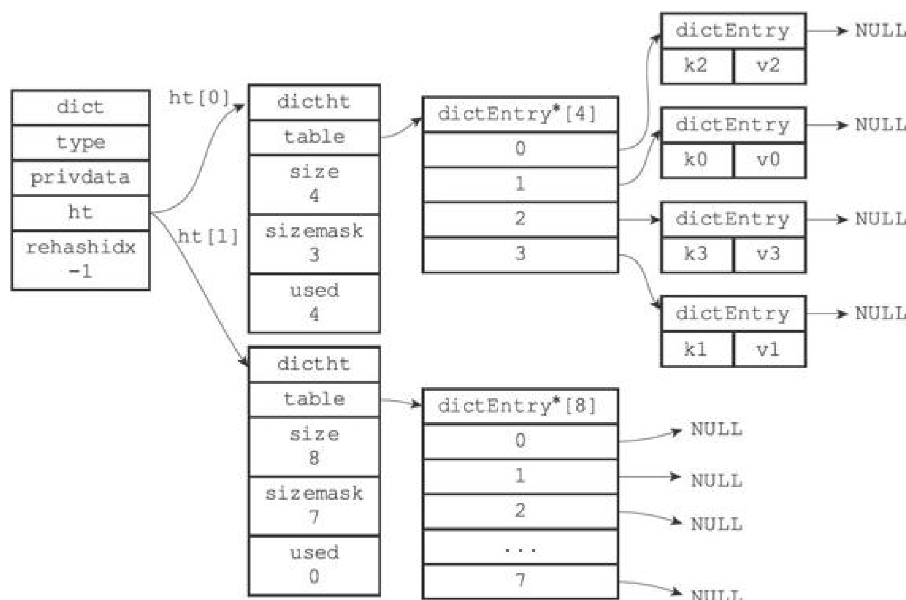


图4-9　为字典的ht[1]哈希表分配空间

2）将ht[0]包含的四个键值对都rehash到ht[1]，如图4-10所示。

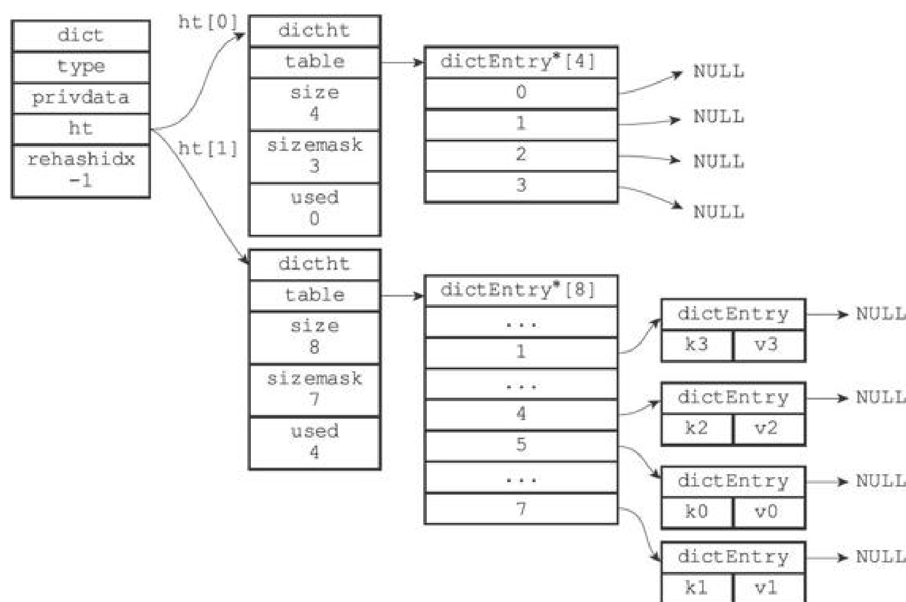


图4-10　ht[0]的所有键值对都已经被迁移到ht[1]

3）释放ht[0]，并将ht[1]设置为ht[0]，然后为ht[1]分配一个空白哈希表，如图4-11所示。至此，对哈希表的扩展操作执行完毕，程序成功将哈希表的大小从原来的4改为了现在的8。

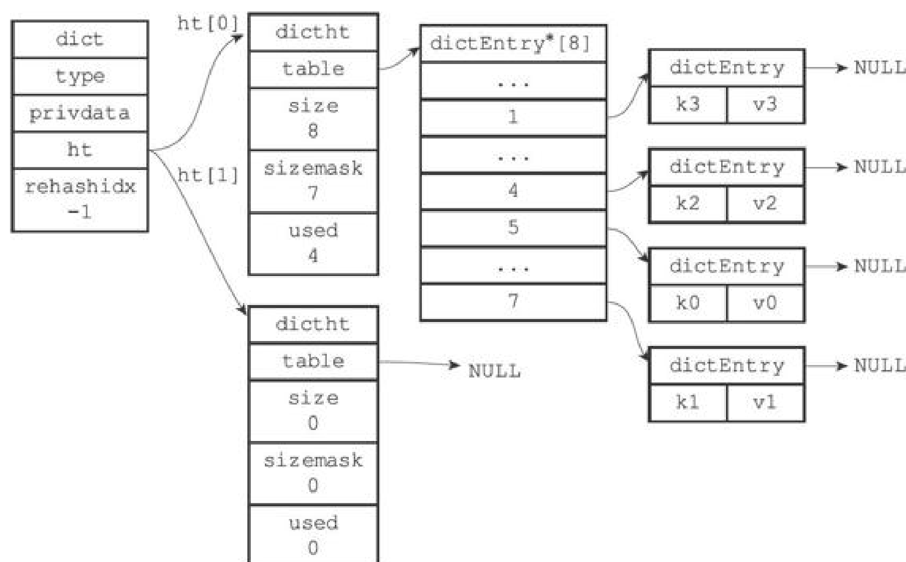


图4-11　完成rehash之后的字典

哈希表的扩展与收缩

当以下条件中的任意一个被满足时，程序会自动开始对哈希表执行扩展操作：

1）服务器目前没有在执行BGSAVE命令或者BGREWRITEAOF命令，并且哈希表的负载因子大于等于1。

2）服务器目前正在执行BGSAVE命令或者BGREWRITEAOF命令，并且哈希表的负载因子大于等于5。

其中哈希表的负载因子可以通过公式：

```
#
负载因子=
哈希表已保存节点数量/
哈希表大小
load_factor = ht[0].used / ht[0].size
```

计算得出。

例如，对于一个大小为4，包含4个键值对的哈希表来说，这个哈希表的负载因子为：

```
load_factor = 4 / 4 = 1
```

又例如，对于一个大小为512，包含256个键值对的哈希表来说，这个哈希表的负载因子为：

```
load_factor = 256 / 512 = 0.5
```

根据BGSAVE命令或BGREWRITEAOF命令是否正在执行，服务器执行扩展操作所需的负载因子并不相同，这是因为在执行BGSAVE命令或BGREWRITEAOF命令的过程中，Redis需要创建当前服务器进程的子进程，而大多数操作系统都采用写时复制（copy-on-write）技术来优化子进程的使用效率，所以在子进程存在期间，服务器会提高执行扩展操作所需的负载因子，从而尽可能地避免在子进程存在期间进行哈希表扩展操作，这可以避免不必要的内存写入操作，最大限度地节约内存。

另一方面，当哈希表的负载因子小于0.1时，程序自动开始对哈希表执行收缩操作。

4.5 渐进式rehash

上一节说过，扩展或收缩哈希表需要将ht[0]里面的所有键值对rehash到ht[1]里面，但是，这个rehash动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的。

这样做的原因在于，如果ht[0]里只保存着四个键值对，那么服务器可以在瞬间就将这些键值对全部rehash到ht[1]；但是，如果哈希表里保存的键值对数量不是四个，而是四百万、四千万甚至四亿个键值对，那么要一次性将这些键值对全部rehash到ht[1]的话，庞大的计算量可能会导致服务器在一段时间内停止服务。

因此，为了避免rehash对服务器性能造成影响，服务器不是一次性将ht[0]里面的所有键值对全部rehash到ht[1]，而是分多次、渐进式地将ht[0]里面的键值对慢慢地rehash到ht[1]。

以下是哈希表渐进式rehash的详细步骤：

1）为ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表。

2）在字典中维持一个索引计数器变量rehashidx，并将它的值设置为0，表示rehash工作正式开始。

3）在rehash进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将ht[0]哈希表在rehashidx索引上的所有键值对rehash到ht[1]，当rehash工作完成之后，程序将rehashidx属性的值增一。

4）随着字典操作的不断执行，最终在某个时间点上，ht[0]的所有键值对都会被rehash至ht[1]，这时程序将rehashidx属性的值设为-1，表示rehash操作已完成。

渐进式rehash的好处在于它采取分而治之的方式，将rehash键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上，从而避免了集中式rehash而带来的庞大计算量。

图4-12至图4-17展示了一次完整的渐进式rehash过程，注意观察在整个rehash过程中，字典的rehashidx属性是如何变化的。

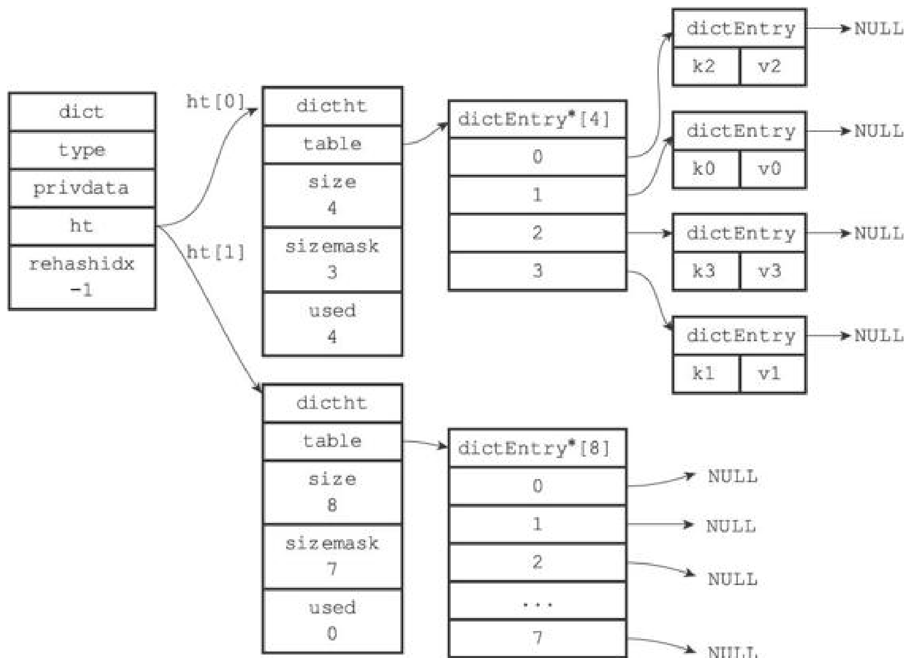


图4-12　准备开始rehash

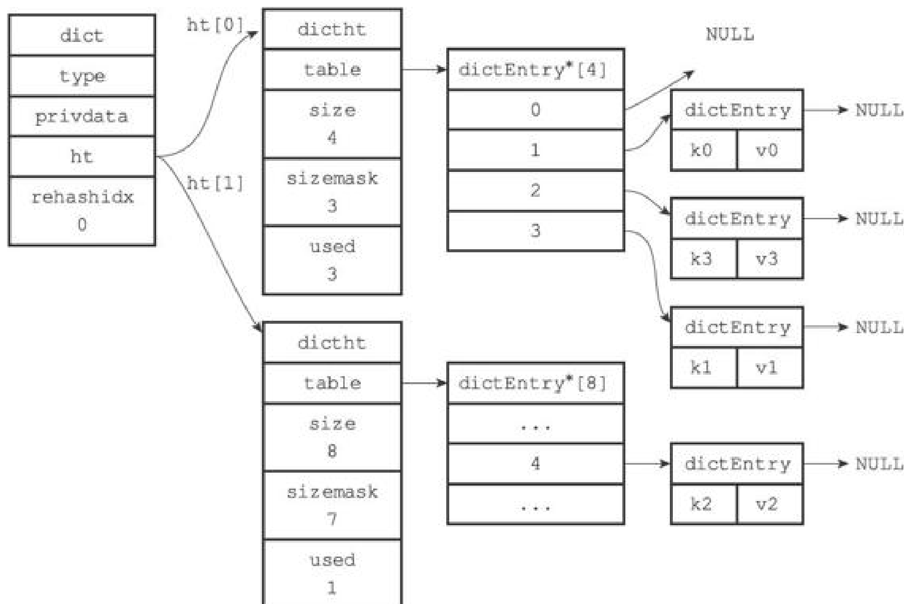


图4-13　rehash索引0上的键值对

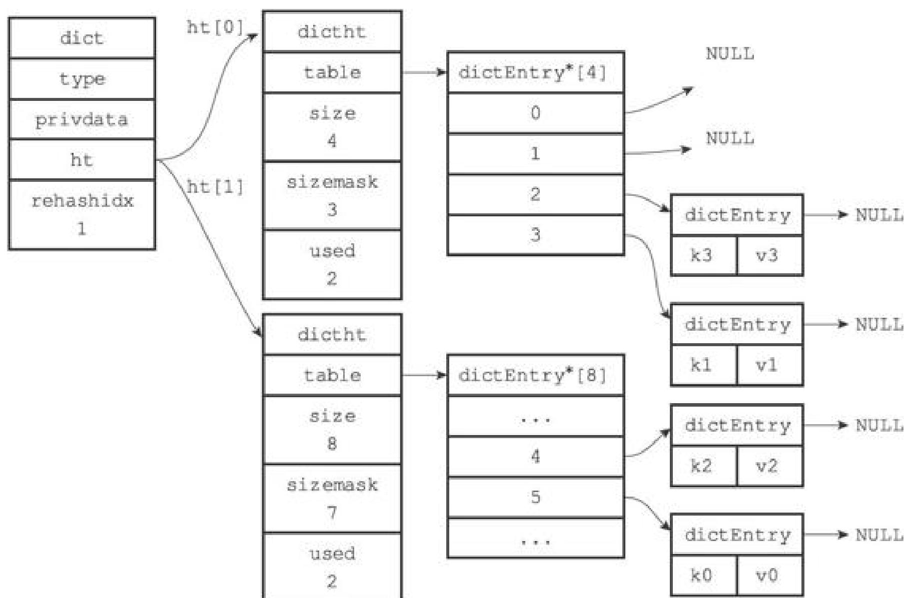


图4-14　rehash索引1上的键值对

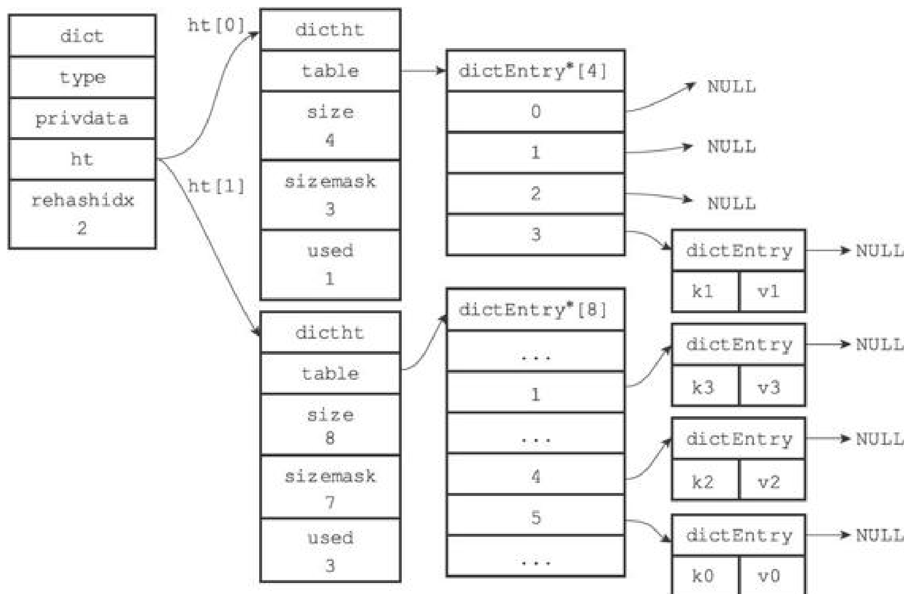


图4-15　rehash索引2上的键值对

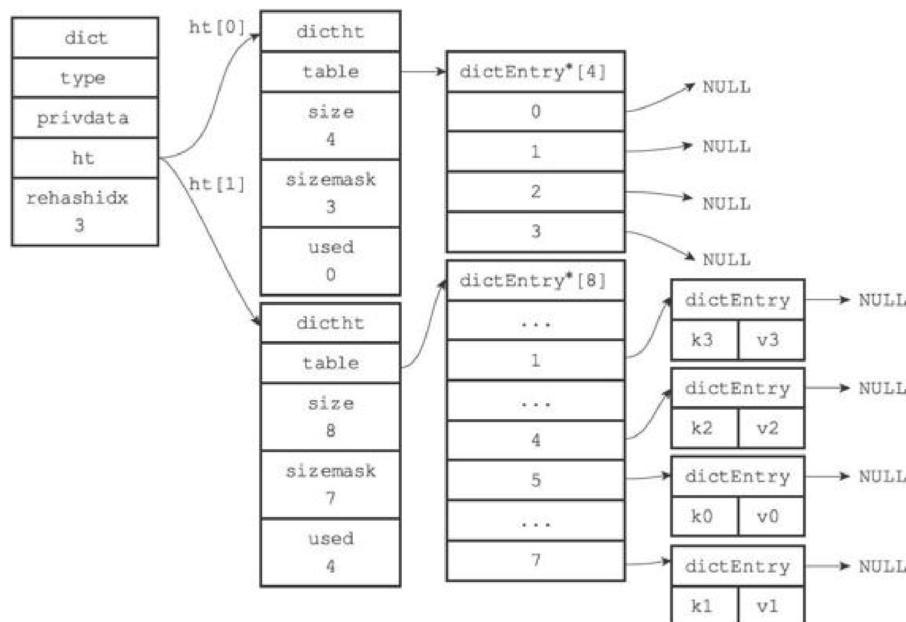


图4-16　rehash索引3上的键值对

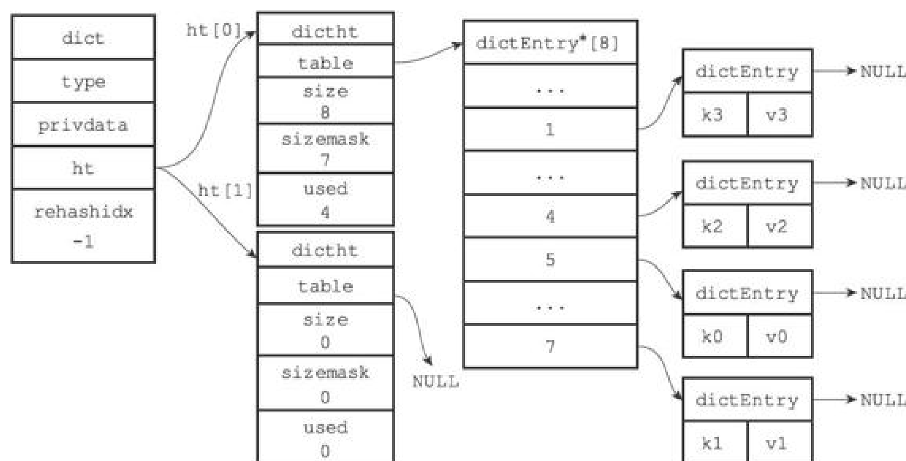


图4-17　rehash执行完毕

渐进式rehash执行期间的哈希表操作

因为在进行渐进式rehash的过程中，字典会同时使用ht[0]和ht[1]两个哈希表，所以在渐进式rehash进行期间，字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表上进行。例如，要在字典里面查找一个键的话，程序会先在ht[0]里面进行查找，如果没找到的话，就会继续到ht[1]里面进行查找，诸如此类。

另外，在渐进式rehash执行期间，新添加到字典的键值对一律会被保存到ht[1]里面，而ht[0]则不再进行任何添加操作，这一措施保证了

ht[0]包含的键值对数量会只减不增，并随着rehash操作的执行而最终变成空表。

4.6 字典API

表4-1列出了字典的主要操作API。

表4-1 字典的主要操作API

函数	作用	时间复杂度
dictCreate	创建一个新的字典	$O(1)$
dictAdd	将给定的键值对添加到字典里面	$O(1)$
dictReplace	将给定的键值对添加到字典里面，如果键已经存在于字典，那么用新值取代原有的值	$O(1)$
dictFetchValue	返回给定键的值	$O(1)$
dictGetRandomKey	从字典中随机返回一个键值对	$O(1)$
dictDelete	从字典中删除给定键所对应的键值对	$O(1)$
dictRelease	释放给定字典，以及字典中包含的所有键值对	$O(N)$，N 为字典包含的键值对数量

4.7 重点回顾

·字典被广泛用于实现Redis的各种功能，其中包括数据库和哈希键。

·Redis中的字典使用哈希表作为底层实现，每个字典带有两个哈希表，一个平时使用，另一个仅在进行rehash时使用。

·当字典被用作数据库的底层实现，或者哈希键的底层实现时，Redis使用MurmurHash2算法来计算键的哈希值。

·哈希表使用链地址法来解决键冲突，被分配到同一个索引上的多个键值对会连接成一个单向链表。

·在对哈希表进行扩展或者收缩操作时，程序需要将现有哈希表包含的所有键值对rehash到新哈希表里面，并且这个rehash过程并不是一次性地完成的，而是渐进式地完成的。

第5章　跳跃表

跳跃表（skiplist）是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

跳跃表支持平均O（logN）、最坏O（N）复杂度的节点查找，还可以通过顺序性操作来批量处理节点。

在大部分情况下，跳跃表的效率可以和平衡树相媲美，并且因为跳跃表的实现比平衡树要来得更为简单，所以有不少程序都使用跳跃表来代替平衡树。

Redis使用跳跃表作为有序集合键的底层实现之一，如果一个有序集合包含的元素数量比较多，又或者有序集合中元素的成员（member）是比较长的字符串时，Redis就会使用跳跃表来作为有序集合键的底层实现。

举个例子，fruit-price是一个有序集合键，这个有序集合以水果名为成员，水果价钱为分值，保存了130款水果的价钱：

```
redis> ZRANGE fruit-price 0 2 WITHSCORES
1)"banana"
2)"5"
3)"cherry"
4)"6.5"
5)"apple"
6)"8"
redis> ZCARD fruit-price
(integer)130
```

fruit-price有序集合的所有数据都保存在一个跳跃表里面，其中每个跳跃表节点（node）都保存了一款水果的价钱信息，所有水果按价钱的高低从低到高在跳跃表里面排序：

·跳跃表的第一个元素的成员为"banana"，它的分值为5；

·跳跃表的第二个元素的成员为"cherry"，它的分值为6.5；

·跳跃表的第三个元素的成员为"apple"，它的分值为8；

和链表、字典等数据结构被广泛地应用在Redis内部不同，Redis只在两个地方用到了跳跃表，一个是实现有序集合键，另一个是在集群节点中用作内部数据结构，除此之外，跳跃表在Redis里面没有其他用途。本章将对Redis中的跳跃表实现进行介绍，并列出跳跃表的操作API。本章不会对跳跃表的基本定义和基础算法进行介绍，如果有需要的话，可以参考WilliamPugh关于跳跃表的论文《Skip Lists:A Probabilistic Alternative to Balanced Trees》，或者《算法：C语言实现（第1～4部分）》一书的13.5节。

5.1 跳跃表的实现

Redis的跳跃表由redis.h/zskiplistNode和redis.h/zskiplist两个结构定义，其中zskiplistNode结构用于表示跳跃表节点，而zskiplist结构则用于保存跳跃表节点的相关信息，比如节点的数量，以及指向表头节点和表尾节点的指针等等。

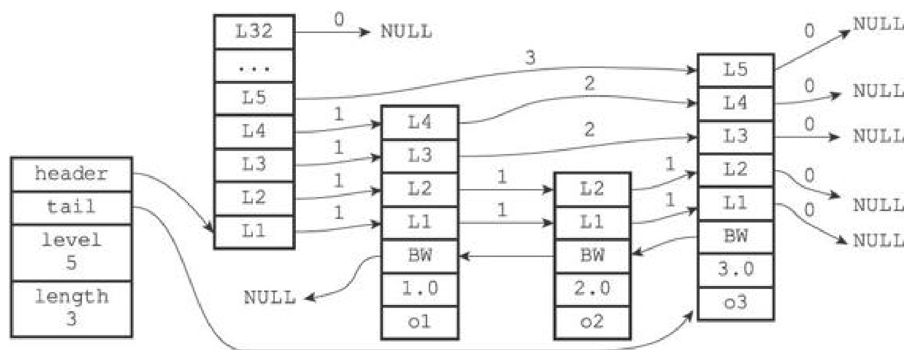


图5-1　一个跳跃表

图5-1展示了一个跳跃表示例，位于图片最左边的是zskiplist结构，该结构包含以下属性：

·header：指向跳跃表的表头节点。

·tail：指向跳跃表的表尾节点。

·level：记录目前跳跃表内，层数最大的那个节点的层数（表头节点的层数不计算在内）。

·length：记录跳跃表的长度，也即是，跳跃表目前包含节点的数量（表头节点不计算在内）。

位于zskiplist结构右方的是四个zskiplistNode结构，该结构包含以下属性：

·层（level）：节点中用L1、L2、L3等字样标记节点的各个层，L1代表第一层，L2代表第二层，以此类推。每个层都带有两个属性：前进指针和跨度。前进指针用于访问位于表尾方向的其他节点，而跨度则记录了前进指针所指向节点和当前节点的距离。在上面的图片中，连线上带有数字的箭头就代表前进指针，而那个数字就是跨度。当程序从表头向表尾进行遍历时，访问会沿着层的前进指针进行。

·后退（backward）指针：节点中用BW字样标记节点的后退指针，它指向位于当前节点的前一个节点。后退指针在程序从表尾向表头遍历时使用。

·分值（score）：各个节点中的1.0、2.0和3.0是节点所保存的分值。在跳跃表中，节点按各自所保存的分值从小到大排列。

·成员对象（obj）：各个节点中的o1、o2和o3是节点所保存的成员对象。

注意表头节点和其他节点的构造是一样的：表头节点也有后退指针、分值和成员对象，不过表头节点的这些属性都不会被用到，所以图中省略了这些部分，只显示了表头节点的各个层。

本节接下来的内容将对zskiplistNode和zskiplist两个结构进行更详细的介绍。

5.1.1 跳跃表节点

跳跃表节点的实现由redis.h/zskiplistNode结构定义：

```
typedef struct zskiplistNode {
    //
层
    struct zskiplistLevel {
        //
前进指针
        struct zskiplistNode *forward;
        //
跨度
        unsigned int span;
    } level[];
    //
后退指针
    struct zskiplistNode *backward;
    //
分值
    double score;
    //
成员对象
    robj *obj;
} zskiplistNode;
```

1.层

跳跃表节点的level数组可以包含多个元素，每个元素都包含一个指向其他节点的指针，程序可以通过这些层来加快访问其他节点的速度，一般来说，层的数量越多，访问其他节点的速度就越快。

每次创建一个新跳跃表节点的时候，程序都根据幂次定律（power law，越大的数出现的概率越小）随机生成一个介于1和32之间的值作为level数组的大小，这个大小就是层的“高度”。

图5-2分别展示了三个高度为1层、3层和5层的节点，因为C语言的数组索引总是从0开始的，所以节点的第一层是level[0]，而第二层是level[1]，以此类推。

2.前进指针

每个层都有一个指向表尾方向的前进指针（level[i].forward属性），用于从表头向表尾方向访问节点。图5-3用虚线表示出了程序从表头向表尾方向，遍历跳跃表中所有节点的路径：

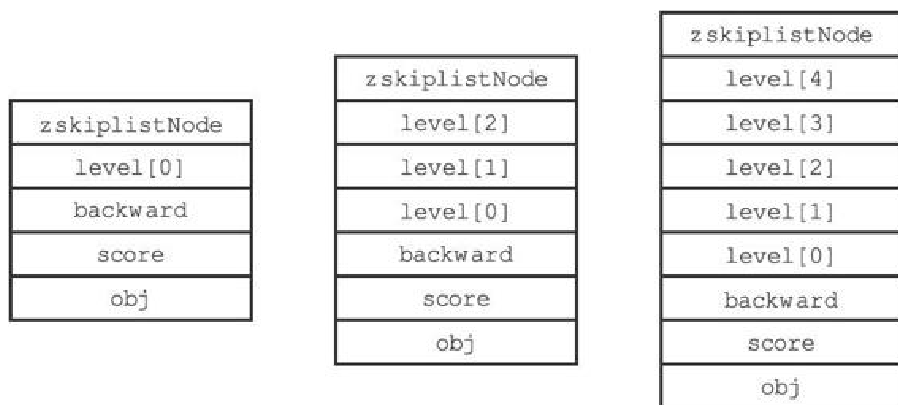


图5-2　带有不同层高的节点

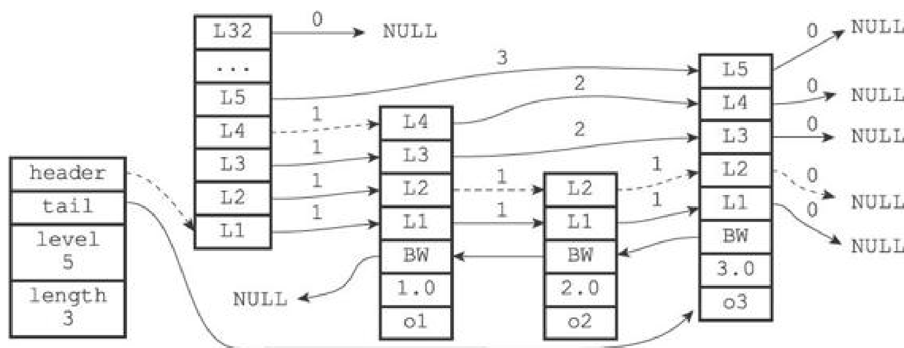


图5-3　遍历整个跳跃表

1）迭代程序首先访问跳跃表的第一个节点（表头），然后从第四层的前进指针移动到表中的第二个节点。

2）在第二个节点时，程序沿着第二层的前进指针移动到表中的第三个节点。

3）在第三个节点时，程序同样沿着第二层的前进指针移动到表中的第四个节点。

4）当程序再次沿着第四个节点的前进指针移动时，它碰到一个NULL，程序知道这时已经到达了跳跃表的表尾，于是结束这次遍历。

3.跨度

层的跨度（level[i].span属性）用于记录两个节点之间的距离：

·两个节点之间的跨度越大，它们相距得就越远。

·指向NULL的所有前进指针的跨度都为0，因为它们没有连向任何节点。

初看上去，很容易以为跨度和遍历操作有关，但实际上并不是这样，遍历操作只使用前进指针就可以完成了，跨度实际上是用来计算排位（rank）的：在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位。

举个例子，图5-4用虚线标记了在跳跃表中查找分值为3.0、成员对象为o3的节点时，沿途经历的层：查找的过程只经过了一个层，并且层的跨度为3，所以目标节点在跳跃表中的排位为3。

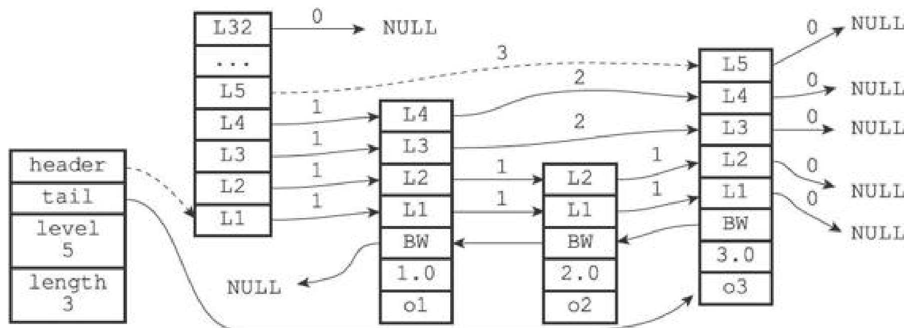


图5-4　计算节点的排位

再举个例子，图5-5用虚线标记了在跳跃表中查找分值为2.0、成员对象为o2的节点时，沿途经历的层：在查找节点的过程中，程序经过了两个跨度为1的节点，因此可以计算出，目标节点在跳跃表中的排位为2。

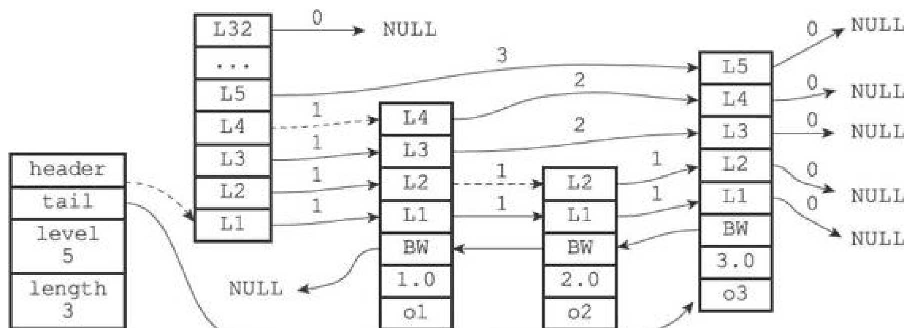


图5-5　另一个计算节点排位的例子

4.后退指针

节点的后退指针（backward属性）用于从表尾向表头方向访问节点：跟可以一次跳过多个节点的前进指针不同，因为每个节点只有一个

后退指针，所以每次只能后退至前一个节点。

图5-6用虚线展示了如果从表尾向表头遍历跳跃表中的所有节点：程序首先通过跳跃表的tail指针访问表尾节点，然后通过后退指针访问倒数第二个节点，之后再沿着后退指针访问倒数第三个节点，再之后遇到指向NULL的后退指针，于是访问结束。

5.分值和成员

节点的分值（score属性）是一个double类型的浮点数，跳跃表中的所有节点都按分值从小到大来排序。

节点的成员对象（obj属性）是一个指针，它指向一个字符串对象，而字符串对象则保存着一个SDS值。

在同一个跳跃表中，各个节点保存的成员对象必须是唯一的，但是多个节点保存的分值却可以是相同的：分值相同的节点将按照成员对象在字典序中的大小来进行排序，成员对象较小的节点会排在前面（靠近表头的方向），而成员对象较大的节点则会排在后面（靠近表尾的方向）。

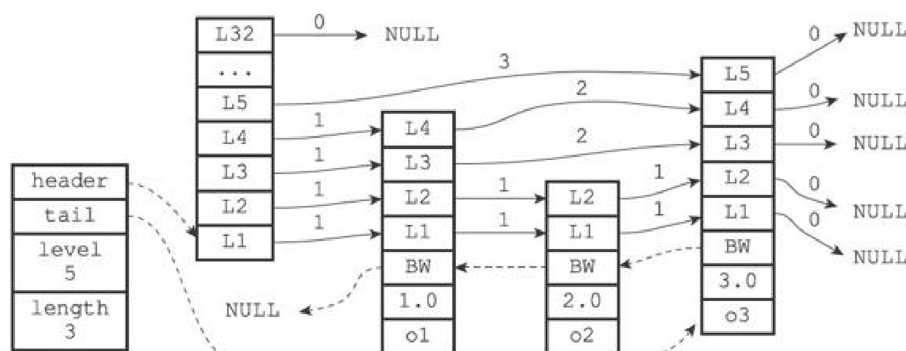


图5-6　从表尾向表头方向遍历跳跃表

举个例子，在图5-7所示的跳跃表中，三个跳跃表节点都保存了相同的分值10086.0，但保存成员对象o1的节点却排在保存成员对象o2和o3的节点之前，而保存成员对象o2的节点又排在保存成员对象o3的节点之前，由此可见，o1、o2、o3三个成员对象在字典中的排序为o1＜＝o2＜＝o3。

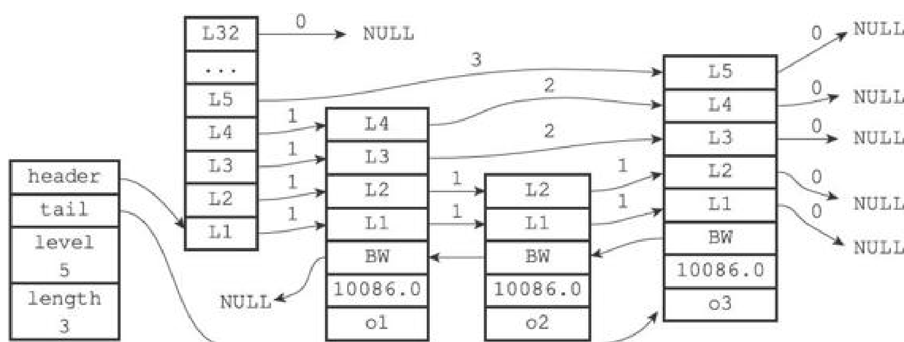


图5-7　三个带有相同分值的跳跃表节点

5.1.2　跳跃表

仅靠多个跳跃表节点就可以组成一个跳跃表，如图5-8所示。

但通过使用一个zskiplist结构来持有这些节点，程序可以更方便地对整个跳跃表进行处理，比如快速访问跳跃表的表头节点和表尾节点，或者快速地获取跳跃表节点的数量（也即是跳跃表的长度）等信息，如图5-9所示。

zskiplist结构的定义如下：

```
typedef struct zskiplist {
    //
表头节点和表尾节点
    structz skiplistNode *header, *tail;
    //
表中节点的数量
    unsigned long length;
    //
表中层数最大的节点的层数
    int level;
} zskiplist;
```

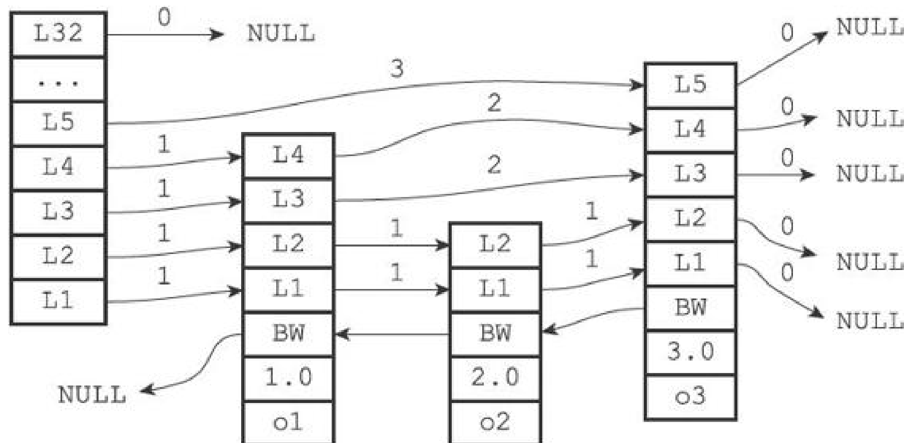


图5-8　由多个跳跃节点组成的跳跃表

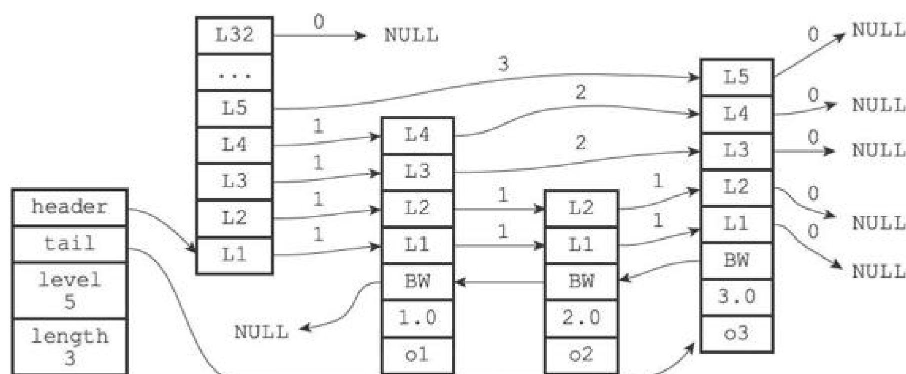


图5-9　带有zskiplist结构的跳跃表

header和tail指针分别指向跳跃表的表头和表尾节点，通过这两个指针，程序定位表头节点和表尾节点的复杂度为O（1）。

通过使用length属性来记录节点的数量，程序可以在O（1）复杂度内返回跳跃表的长度。

level属性则用于在O（1）复杂度内获取跳跃表中层高最大的那个节点的层数量，注意表头节点的层高并不计算在内。

5.2 跳跃表API

表5-1列出了跳跃表的所有操作API。

表5-1 跳跃表API

函数	作用	时间复杂度
zslCreate	创建一个新的跳跃表	$O(1)$
zslFree	释放给定跳跃表，以及表中包含的所有节点	$O(N)$，N 为跳跃表的长度
zslInsert	将包含给定成员和分值的新节点添加到跳跃表中	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslDelete	删除跳跃表中包含给定成员和分值的节点	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslGetRank	返回包含给定成员和分值的节点在跳跃表中的排位	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslGetElementByRank	返回跳跃表在给定排位上的节点	平均 $O(\log N)$，最坏 $O(N)$，N 为跳跃表长度
zslIsInRange	给定一个分值范围（range），比如 0 到 15，20 到 28，诸如此类，如果跳跃表中有至少一个节点的分值在这个范围之内，那么返回 1，否则返回 0	通过跳跃表的表头节点和表尾节点，这个检测可以用 $O(1)$ 复杂度完成
zslFirstInRange	给定一个分值范围，返回跳跃表中第一个符合这个范围的节点	平均 $O(\log N)$，最坏 $O(N)$。N 为跳跃表长度
zslLastInRange	给定一个分值范围，返回跳跃表中最后一个符合这个范围的节点	平均 $O(\log N)$，最坏 $O(N)$。N 为跳跃表长度
zslDeleteRangeByScore	给定一个分值范围，删除跳跃表中所有在这个范围之内的节点	$O(N)$，N 为被删除节点数量
zslDeleteRangeByRank	给定一个排位范围，删除跳跃表中所有在这个范围之内的节点	$O(N)$，N 为被删除节点数量

5.3 重点回顾

·跳跃表是有序集合的底层实现之一。

·Redis的跳跃表实现由zskiplist和zskiplistNode两个结构组成，其中zskiplist用于保存跳跃表信息（比如表头节点、表尾节点、长度），而zskiplistNode则用于表示跳跃表节点。

·每个跳跃表节点的层高都是1至32之间的随机数。

·在同一个跳跃表中，多个节点可以包含相同的分值，但每个节点的成员对象必须是唯一的。

·跳跃表中的节点按照分值大小进行排序，当分值相同时，节点按照成员对象的大小进行排序。

第6章　整数集合

整数集合（intset）是集合键的底层实现之一，当一个集合只包含整数值元素，并且这个集合的元素数量不多时，Redis就会使用整数集合作为集合键的底层实现。

举个例子，如果我们创建一个只包含五个元素的集合键，并且集合中的所有元素都是整数值，那么这个集合键的底层实现就会是整数集合：

```
redis> SADD numbers 1 3 5 7 9
(integer) 5
redis> OBJECT ENCODING numbers
"intset"
```

在这一章，我们将对整数集合及其相关操作的实现原理进行介绍。

6.1 整数集合的实现

整数集合（intset）是Redis用于保存整数值的集合抽象数据结构，它可以保存类型为int16_t、int32_t或者int64_t的整数值，并且保证集合中不会出现重复元素。

每个intset.h/intset结构表示一个整数集合：

```
typedef struct intset {
    //
编码方式
    uint32_t encoding;
    //
集合包含的元素数量
    uint32_t length;
    //
保存元素的数组
    int8_t contents[];
} intset;
```

contents数组是整数集合的底层实现：整数集合的每个元素都是contents数组的一个数组项（item），各个项在数组中按值的大小从小到大有序地排列，并且数组中不包含任何重复项。

length属性记录了整数集合包含的元素数量，也即是contents数组的长度。

虽然intset结构将contents属性声明为int8_t类型的数组，但实际上contents数组并不保存任何int8_t类型的值，contents数组的真正类型取决于encoding属性的值：

·如果encoding属性的值为INTSET_ENC_INT16，那么contents就是一个int16_t类型的数组，数组里的每个项都是一个int16_t类型的整数值（最小值为-32768，最大值为32767）。

·如果encoding属性的值为INTSET_ENC_INT32，那么contents就是一个int32_t类型的数组，数组里的每个项都是一个int32_t类型的整数值（最小值为-2147483648，最大值为2147483647）。

·如果encoding属性的值为INTSET_ENC_INT64，那么contents就是一个int64_t类型的数组，数组里的每个项都是一个int64_t类型的整数值（最小值为-9223372036854775808，最大值为9223372036854775807）。

图6-1展示了一个整数集合示例：

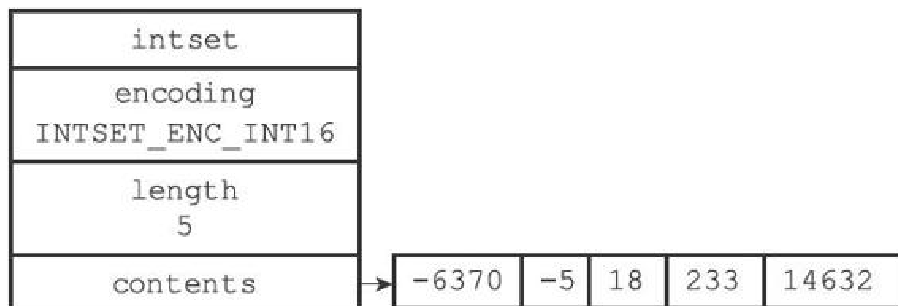


图6-1　一个包含五个int16_t类型整数值的整数集合

·encoding属性的值为INTSET_ENC_INT16，表示整数集合的底层实现为int16_t类型的数组，而集合保存的都是int16_t类型的整数值。

·length属性的值为5，表示整数集合包含五个元素。

·contents数组按从小到大的顺序保存着集合中的五个元素。

·因为每个集合元素都是int16_t类型的整数值，所以contents数组的大小等于sizeof（int16_t）*5＝16*5＝80位。

图6-2展示了另一个整数集合示例：

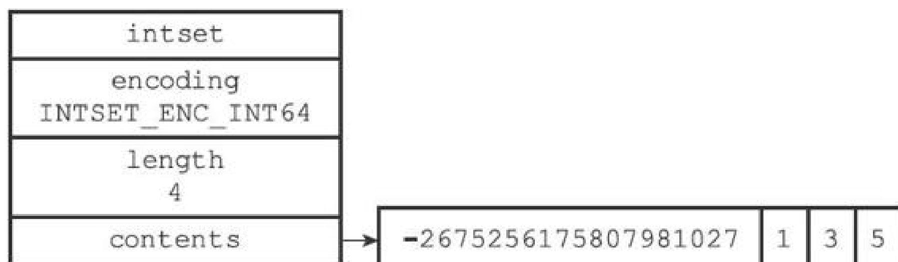


图6-2　一个包含四个int16_t类型整数值的整数集合

·encoding属性的值为INTSET_ENC_INT64，表示整数集合的底层实现为int64_t类型的数组，而数组中保存的都是int64_t类型的整数值。

·length属性的值为4，表示整数集合包含四个元素。

·contents数组按从小到大的顺序保存着集合中的四个元素。

·因为每个集合元素都是int64_t类型的整数值，所以contents数组的大小为sizeof（int64_t）*4＝64*4＝256位。

虽然contents数组保存的四个整数值中，只有-2675256175807981027是真正需要用int64_t类型来保存的，而其他的1、3、5三个值都可以用int16_t类型来保存，不过根据整数集合的升

级规则，当向一个底层为int16_t数组的整数集合添加一个int64_t类型的整数值时，整数集合已有的所有元素都会被转换成int64_t类型，所以contents数组保存的四个整数值都是int64_t类型的，不仅仅是-2675256175807981027。

接下来的一节将对整数集合的升级操作进行详细介绍。

6.2 升级

每当我们要将一个新元素添加到整数集合里面，并且新元素的类型比整数集合现有所有元素的类型都要长时，整数集合需要先进行升级（upgrade），然后才能将新元素添加到整数集合里面。

升级整数集合并添加新元素共分为三步进行：

1）根据新元素的类型，扩展整数集合底层数组的空间大小，并为新元素分配空间。

2）将底层数组现有的所有元素都转换成与新元素相同的类型，并将类型转换后的元素放置到正确的位上，而且在放置元素的过程中，需要继续维持底层数组的有序性质不变。

3）将新元素添加到底层数组里面。

举个例子，假设现在有一个INTSET_ENC_INT16编码的整数集合，集合中包含三个int16_t类型的元素，如图6-3所示。

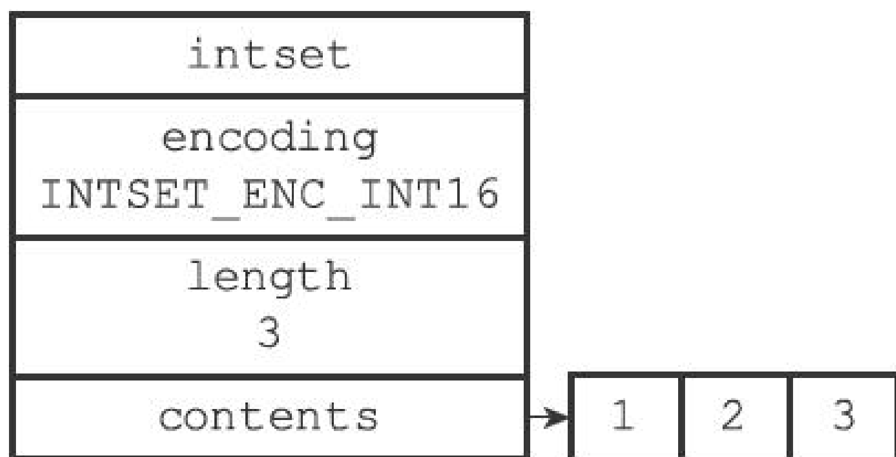


图6-3　一个包含三个int16_t类型的元素的整数集合

因为每个元素都占用16位空间，所以整数集合底层数组的大小为3*16＝48位，图6-4展示了整数集合的三个元素在这48位里的位置。

位	0至15位	16至31位	32至47位
元素	1	2	3

图6-4　contents数组的各个元素，以及它们所在的位

现在，假设我们要将类型为int32_t的整数值65535添加到整数集合里面，因为65535的类型int32_t比整数集合当前所有元素的类型都要长，所以在将65535添加到整数集合之前，程序需要先对整数集合进行升级。

升级首先要做的是，根据新类型的长度，以及集合元素的数量（包括要添加的新元素在内），对底层数组进行空间重分配。

整数集合目前有三个元素，再加上新元素65535，整数集合需要分配四个元素的空间，因为每个int32_t整数值需要占用32位空间，所以在空间重分配之后，底层数组的大小将是32*4＝128位，如图6-5所示。虽然程序对底层数组进行了空间重分配，但数组原有的三个元素1、2、3仍然是int16_t类型，这些元素还保存在数组的前48位里面，所以程序接下来要做的就是将这三个元素转换成int32_t类型，并将转换后的元素放置到正确的位上面，而且在放置元素的过程中，需要维持底层数组的有序性质不变。

位	0至15位	16至31位	32至47位	48至127位
元素	1	2	3	（新分配空间）

图6-5　进行空间重分配之后的数组

首先，因为元素3在1、2、3、65535四个元素中排名第三，所以它将被移动到contents数组的索引2位置上，也即是数组64位至95位的空间内，如图6-6所示。

位	0至15位	16至31位	32至47位	48至63位	64位至95位	96位至127位
元素	1	2	3	（新分配空间）	3	（新分配空间）

从int16_t类型转换为int32_t类型

图6-6　对元素3进行类型转换，并保存在适当的位上

接着，因为元素2在1、2、3、65535四个元素中排名第二，所以它将被移动到contents数组的索引1位置上，也即是数组的32位至63位的空间内，如图6-7所示。

位	0至15位	16至31位	32至63位	64位至95位	96位至127位
元素	1	2	2	3	（新分配空间）

从int16_t类型转换为int32_t类型

图6-7　对元素2进行类型转换，并保存在适当的位上

之后，因为元素1在1、2、3、65535四个元素中排名第一，所以它将被移动到contents数组的索引0位置上，即数组的0位至31位的空间内，如图6-8所示。

位	0至31位	32至63位	64位至95位	96位至127位
元素	1	2	3	（新分配空间）

从 int16_t 类型转换为 int32_t 类型

图6-8　对元素1进行类型转换，并保存在适当的位上

然后，因为元素65535在1、2、3、65535四个元素中排名第四，所以它将被添加到contents数组的索引3位置上，也即是数组的96位至127位的空间内，如图6-9所示。

位	0至31位	32至63位	64位至95位	96位至127位
元素	1	2	3	65535

添加新元素

图6-9　添加65535到数组

最后，程序将整数集合encoding属性的值从INTSET_ENC_INT16改为INTSET_ENC_INT32，并将length属性的值从3改为4，设置完成之后的整数集合如图6-10所示。



图6-10　完成添加操作之后的整数集合

因为每次向整数集合添加新元素都可能会引起升级，而每次升级都需要对底层数组中已有的所有元素进行类型转换，所以向整数集合添加新元素的时间复杂度为O（N）。

其他类型的升级操作，比如从INTSET_ENC_INT16编码升级为INTSET_ENC_INT64编码，或者从INTSET_ENC_INT32编码升级为INTSET_ENC_INT64编码，升级的过程都和上面展示的升级过程类似。

升级之后新元素的摆放位置

因为引发升级的新元素的长度总是比整数集合现有所有元素的长度都大，所以这个新元素的值要么就大于所有现有元素，要么就小于所有现有元素：

·在新元素小于所有现有元素的情况下，新元素会被放置在底层数组的最开头（索引0）；

·在新元素大于所有现有元素的情况下，新元素会被放置在底层数组的最末尾（索引length-1）。

6.3 升级的好处

整数集合的升级策略有两个好处，一个是提升整数集合的灵活性，另一个是尽可能地节约内存。

6.3.1 提升灵活性

因为C语言是静态类型语言，为了避免类型错误，我们通常不会将两种不同类型的值放在同一个数据结构里面。

例如，我们一般只使用int16_t类型的数组来保存int16_t类型的值，只使用int32_t类型的数组来保存int32_t类型的值，诸如此类。

但是，因为整数集合可以通过自动升级底层数组来适应新元素，所以我们可以随意地将int16_t、int32_t或者int64_t类型的整数添加到集合中，而不必担心出现类型错误，这种做法非常灵活。

6.3.2 节约内存

当然，要让一个数组可以同时保存int16_t、int32_t、int64_t三种类型的值，最简单的做法就是直接使用int64_t类型的数组作为整数集合的底层实现。不过这样一来，即使添加到整数集合里面的都是int16_t类型或者int32_t类型的值，数组都需要使用int64_t类型的空间去保存它们，从而出现浪费内存的情况。

而整数集合现在的做法既可以让集合能同时保存三种不同类型的值，又可以确保升级操作只会在有需要的时候进行，这可以尽量节省内存。

例如，如果我们一直只向整数集合添加int16_t类型的值，那么整数集合的底层实现就会一直是int16_t类型的数组，只有在我们要将int32_t类型或者int64_t类型的值添加到集合时，程序才会对数组进行升级。

6.4 降级

整数集合不支持降级操作，一旦对数组进行了升级，编码就会一直保持升级后的状态。

举个例子，对于图6-11所示的整数集合来说，即使我们将集合里唯一一个真正需要使用int64_t类型来保存的元素4294967295删除了，整数集合的编码仍然会维持INTSET_ENC_INT64，底层数组也仍然会是int64_t类型的，如图6-12所示。

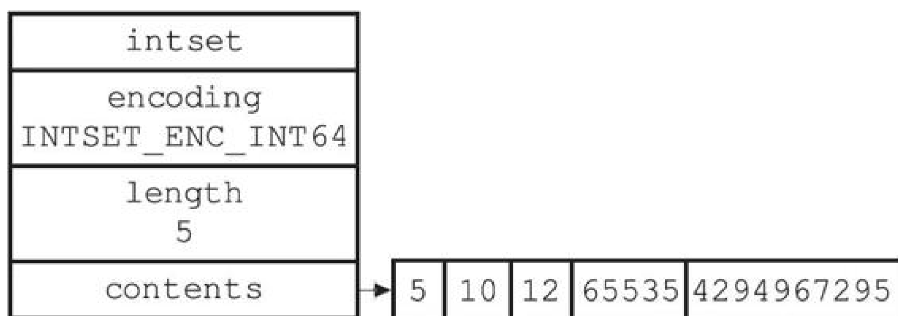


图6-11　数组编码为INTSET_ENC_INT64的整数集合

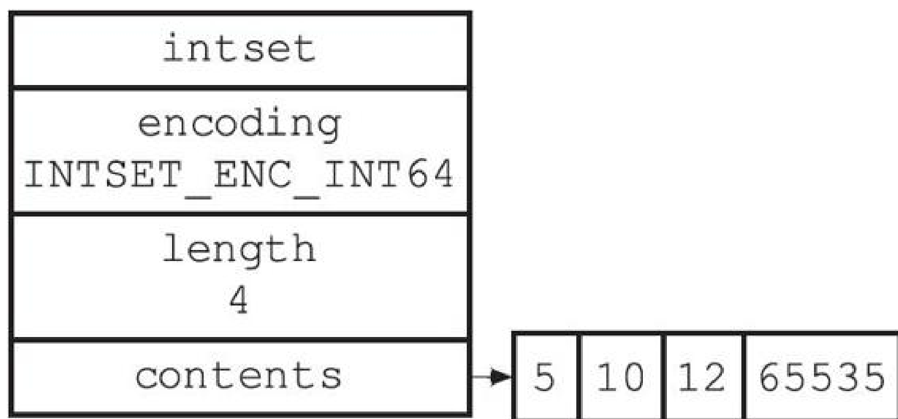


图6-12　删除4 294 967 295的整数集合

6.5 整数集合API

表6-1列出了整数集合的操作API。

表6-1 整数集合API

函 数	作 用	时间复杂度
intsetNew	创建一个新的压缩列表	$O(1)$
intsetAdd	将给定元素添加到整数集合里面	$O(N)$
intsetRemove	从整数集合中移除给定元素	$O(N)$
intsetFind	检查给定值是否存在于集合	因为底层数组有序，查找可以通过二分查找法来进行，所以复杂度为 $O(\log N)$
intsetRandom	从整数集合中随机返回一个元素	$O(1)$
intsetGet	取出底层数组在给定索引上的元素	$O(1)$
intsetLen	返回整数集合包含的元素个数	$O(1)$
intsetBlobLen	返回整数集合占用的内存字节数	$O(1)$

6.6 重点回顾

·整数集合是集合键的底层实现之一。

·整数集合的底层实现为数组，这个数组以有序、无重复的方式保存集合元素，在有需要时，程序会根据新添加元素的类型，改变这个数组的类型。

·升级操作为整数集合带来了操作上的灵活性，并且尽可能地节约了内存。

·整数集合只支持升级操作，不支持降级操作。

第7章　压缩列表

压缩列表（ziplist）是列表键和哈希键的底层实现之一。当一个列表键只包含少量列表项，并且每个列表项要么就是小整数值，要么就是长度比较短的字符串，那么Redis就会使用压缩列表来做列表键的底层实现。

例如，执行以下命令将创建一个压缩列表实现的列表键：

```
redis> RPUSH lst 1 3 5 10086 "hello" "world"
(integer)6
redis> OBJECT ENCODING lst
"ziplist"
```

列表键里面包含的都是1、3、5、10086这样的小整数值，以及"hello"、"world"这样的短字符串。

另外，当一个哈希键只包含少量键值对，比且每个键值对的键和值要么就是小整数值，要么就是长度比较短的字符串，那么Redis就会使用压缩列表来做哈希键的底层实现。

举个例子，执行以下命令将创建一个压缩列表实现的哈希键：

```
redis> HMSET profile "name" "Jack" "age" 28 "job" "Programmer"
OK
redis> OBJECT ENCODING profile
"ziplist"
```

哈希键里面包含的所有键和值都是小整数值或者短字符串。本章将对压缩列表的定义以及相关操作进行详细的介绍。

7.1 压缩列表的构成

压缩列表是Redis为了节约内存而开发的，是由一系列特殊编码的连续内存块组成的顺序型（sequential）数据结构。一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。

图7-1展示了压缩列表的各个组成部分，表7-1则记录了各个组成部分的类型、长度以及用途。

zlbytes	zltail	zllen	entry1	entry2	...	entryN	zlend

图7-1 压缩列表的各个组成部分

表7-1 压缩列表各个组成部分的详细说明

属性	类型	长度	用 途
zlbytes	uint32_t	4字节	记录整个压缩列表占用的内存字节数：在对压缩列表进行内存重分配，或者计算 zlend 的位置时使用
zltail	uint32_t	4字节	记录压缩列表表尾节点距离压缩列表的起始地址有多少字节：通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址
zllen	uint16_t	2字节	记录了压缩列表包含的节点数量：当这个属性的值小于 UINT16_MAX（65535）时，这个属性的值就是压缩列表包含节点的数量；当这个值等于 UINT16_MAX 时，节点的真实数量需要遍历整个压缩列表才能计算得出
entryX	列表节点	不定	压缩列表包含的各个节点，节点的长度由节点保存的内容决定
zlend	uint8_t	1字节	特殊值 0xFF（十进制 255），用于标记压缩列表的末端

图7-2展示了一个压缩列表示例：

·列表zlbytes属性的值为0x50（十进制80），表示压缩列表的总长为80字节。

·列表zltail属性的值为0x3c（十进制60），这表示如果我们有一个指向压缩列表起始地址的指针p，那么只要用指针p加上偏移量60，就可以计算出表尾节点entry3的地址。

·列表zllen属性的值为0x3（十进制3），表示压缩列表包含三个节点。



图7-2 包含三个节点的压缩列表

图7-3展示了另一个压缩列表示例：



图7-3　包含五个节点的压缩列表

·列表zlbytes属性的值为0xd2（十进制210），表示压缩列表的总长为210字节。

·列表zltail属性的值为0xb3（十进制179），这表示如果我们有一个指向压缩列表起始地址的指针p，那么只要用指针p加上偏移量179，就可以计算出表尾节点entry5的地址。

·列表zllen属性的值为0x5（十进制5），表示压缩列表包含五个节点。

7.2　压缩列表节点的构成

每个压缩列表节点可以保存一个字节数组或者一个整数值，其中，字节数组可以是以下三种长度的其中一种：

·长度小于等于63（2^{6}–1）字节的字节数组；

·长度小于等于16383（2^{14}–1）字节的字节数组；

·长度小于等于4294967295（2^{32}–1）字节的字节数组；

而整数值则可以是以下六种长度的其中一种：

·4位长，介于0至12之间的无符号整数；

·1字节长的有符号整数；

·3字节长的有符号整数；

·int16_t类型整数；

·int32_t类型整数；

·int64_t类型整数。

每个压缩列表节点都由previous_entry_length、encoding、content三个部分组成，如图7-4所示。



图7-4　压缩列表节点的各个组成部分

接下来的内容将分别介绍这三个组成部分。

7.2.1　previous_entry_length

节点的previous_entry_length属性以字节为单位，记录了压缩列表中前一个节点的长度。previous_entry_length属性的长度可以是1字节或者5字节：

·如果前一节点的长度小于254字节，那么previous_entry_length属性的长度为1字节：前一节点的长度就保存在这一个字节里面。

·如果前一节点的长度大于等于254字节，那么previous_entry_length属性的长度为5字节：其中属性的第一字节会被设置为0xFE（十进制值254），而之后的四个字节则用于保存前一节点的长度。

图7-5展示了一个包含一字节长previous_entry_length属性的压缩列表节点，属性的值为0x05，表示前一节点的长度为5字节。

previous_entry_length 0x05	encoding ...	content ...

图7-5　当前节点的前一节点的长度为5字节

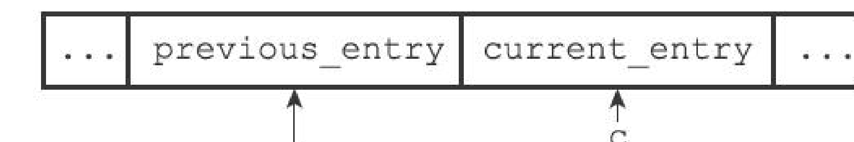
图7-6展示了一个包含五字节长previous_entry_length属性的压缩节点，属性的值为0xFE00002766，其中值的最高位字节0xFE表示这是一个五字节长的previous_entry_length属性，而之后的四字节0x00002766（十进制值10086）才是前一节点的实际长度。

previous_entry_length 0xFE00002766	encoding ...	content ...

图7-6　当前节点的前一节点的长度为10086字节

因为节点的previous_entry_length属性记录了前一个节点的长度，所以程序可以通过指针运算，根据当前节点的起始地址来计算出前一个节点的起始地址。

举个例子，如果我们有一个指向当前节点起始地址的指针c，那么我们只要用指针c减去当前节点previous_entry_length属性的值，就可以得出一个指向前一个节点起始地址的指针p，如图7-7所示。



p = c - current_entry.previous_entry_length

图7-7　通过指针运算计算出前一个节点的地址

压缩列表的从表尾向表头遍历操作就是使用这一原理实现的，只要我们拥有了一个指向某个节点起始地址的指针，那么通过这个指针以及这个节点的previous_entry_length属性，程序就可以一直向前一个节点回溯，最终到达压缩列表的表头节点。

图7-8展示了一个从表尾节点向表头节点进行遍历的完整过程：

·首先，我们拥有指向压缩列表表尾节点entry4起始地址的指针p1（指向表尾节点的指针可以通过指向压缩列表起始地址的指针加上zltail属性的值得出）；

·通过用p1减去entry4节点previous_entry_length属性的值，我们得

到一个指向entry4前一节点entry3起始地址的指针p2；

·通过用p2减去entry3节点previous_entry_length属性的值，我们得到一个指向entry3前一节点entry2起始地址的指针p3；

·通过用p3减去entry2节点previous_entry_length属性的值，我们得到一个指向entry2前一节点entry1起始地址的指针p4，entry1为压缩列表的表头节点；

·最终，我们从表尾节点向表头节点遍历了整个列表。

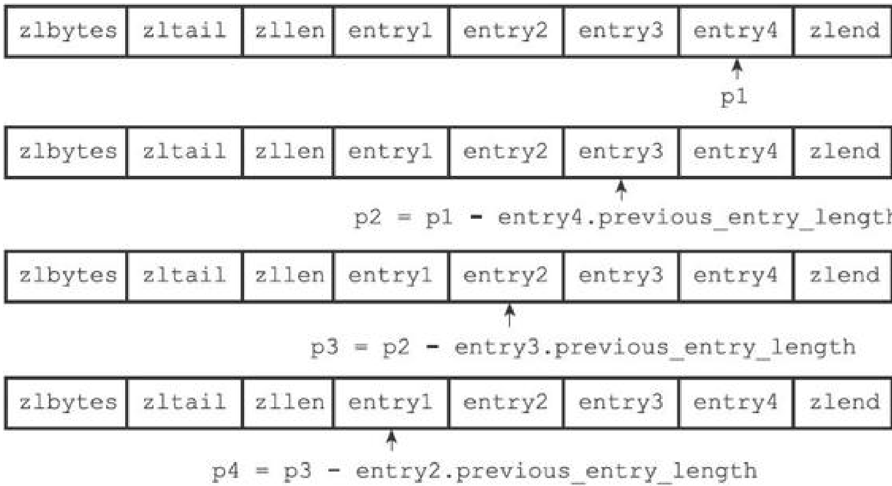


图7-8　一个从表尾向表头遍历的例子

7.2.2　encoding

节点的encoding属性记录了节点的content属性所保存数据的类型以及长度：

·一字节、两字节或者五字节长，值的最高位为00、01或者10的是字节数组编码：这种编码表示节点的content属性保存着字节数组，数组的长度由编码除去最高两位之后的其他位记录；

·一字节长，值的最高位以11开头的是整数编码：这种编码表示节点的content属性保存着整数值，整数值的类型和长度由编码除去最高两位之后的其他位记录；

表7-2记录了所有可用的字节数组编码，而表7-3则记录了所有可用的整数编码。表格中的下划线“_”表示留空，而b、x等变量则代表实际的二进制数据，为了方便阅读，多个字节之间用空格隔开。

表7-2　字节数组编码

编　　码	编码长度	content 属性保存的值
00bbbbbb	1字节	长度小于等于63字节的字节数组
01bbbbbb xxxxxxxx	2字节	长度小于等于16 383字节的字节数组
10______ aaaaaaaa bbbbbbbb cccccccc dddddddd	5字节	长度小于等于4 294 967 295的字节数组

表7-3　整数编码

编码	编码长度	content 属性保存的值
11000000	1字节	int16_t 类型的整数
11010000	1字节	int32_t 类型的整数
11100000	1字节	int64_t 类型的整数
11110000	1字节	24位有符号整数
11111110	1字节	8位有符号整数
1111xxxx	1字节	使用这一编码的节点没有相应的 content 属性，因为编码本身的 xxxx 四个位已经保存了一个介于0和12之间的值，所以它无须 content 属性

7.2.3　content

节点的content属性负责保存节点的值，节点值可以是一个字节数组或者整数，值的类型和长度由节点的encoding属性决定。

图7-9展示了一个保存字节数组的节点示例：

·编码的最高两位00表示节点保存的是一个字节数组；

·编码的后六位001011记录了字节数组的长度11；

·content属性保存着节点的值"hello world"。

previous_entry_length ...	encoding 00001011	content "hello world"

图7-9　保存着节数组"hello world"的节点

图7-10展示了一个保存整数值的节点示例：

previous_entry_length ...	encoding 11000000	content 10086

图7-10　保存着整数值10086的节点

·编码11000000表示节点保存的是一个int16_t类型的整数值；

·content属性保存着节点的值10086。

7.3 连锁更新

前面说过，每个节点的previous_entry_length属性都记录了前一个节点的长度：

·如果前一节点的长度小于254字节，那么previous_entry_length属性需要用1字节长的空间来保存这个长度值。

·如果前一节点的长度大于等于254字节，那么previous_entry_length属性需要用5字节长的空间来保存这个长度值。

现在，考虑这样一种情况：在一个压缩列表中，有多个连续的、长度介于250字节到253字节之间的节点e1至eN，如图7-11所示。



图7-11　包含节点el至eN的压缩列表

因为e1至eN的所有节点的长度都小于254字节，所以记录这些节点的长度只需要1字节长的previous_entry_length属性，换句话说，e1至eN的所有节点的previous_entry_length属性都是1字节长的。

这时，如果我们将一个长度大于等于254字节的新节点new设置为压缩列表的表头节点，那么new将成为e1的前置节点，如图7-12所示。

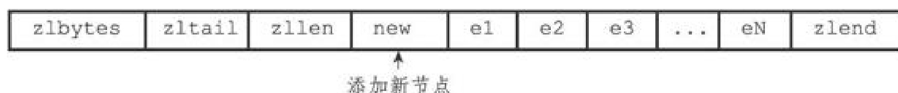


图7-12　添加新节点到压缩列表

因为e1的previous_entry_length属性仅长1字节，它没办法保存新节点new的长度，所以程序将对压缩列表执行空间重分配操作，并将e1节点的previous_entry_length属性从原来的1字节长扩展为5字节长。

现在，麻烦的事情来了，e1原本的长度介于250字节至253字节之间，在为previous_entry_length属性新增四个字节的空间之后，e1的长度就变成了介于254字节至257字节之间，而这种长度使用1字节长的previous_entry_length属性是没办法保存的。

因此，为了让e2的previous_entry_length属性可以记录下e1的长度，程序需要再次对压缩列表执行空间重分配操作，并将e2节点的previous_entry_length属性从原来的1字节长扩展为5字节长。

正如扩展e1引发了对e2的扩展一样，扩展e2也会引发对e3的扩展，而扩展e3又会引发对e4的扩展……为了让每个节点的previous_entry_length属性都符合压缩列表对节点的要求，程序需要不

断地对压缩列表执行空间重分配操作，直到eN为止。

Redis将这种在特殊情况下产生的连续多次空间扩展操作称之为“连锁更新”（cascade update），图7-13展示了这一过程。

除了添加新节点可能会引发连锁更新之外，删除节点也可能会引发连锁更新。

考虑图7-14所示的压缩列表，如果e1至eN都是大小介于250字节至253字节的节点，big节点的长度大于等于254字节（需要5字节的previous_entry_length来保存），而small节点的长度小于254字节（只需要1字节的previous_entry_length来保存），那么当我们将small节点从压缩列表中删除之后，为了让e1的previous_entry_length属性可以记录big节点的长度，程序将扩展e1的空间，并由此引发之后的连锁更新。

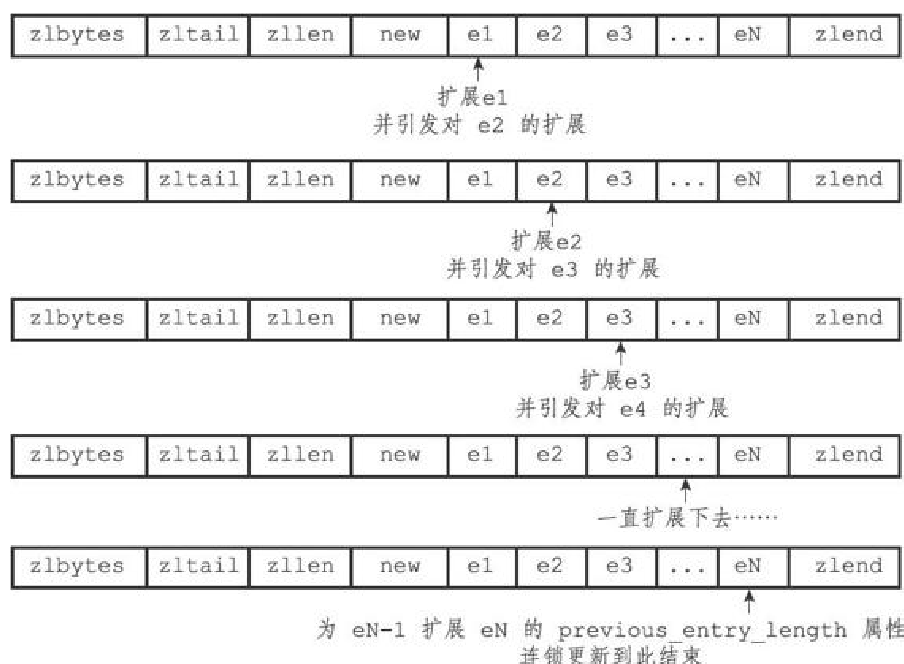


图7-13　连锁更新过程



图7-14　另一种引起连锁更新的情况

因为连锁更新在最坏情况下需要对压缩列表执行N次空间重分配操作，而每次空间重分配的最坏复杂度为O（N），所以连锁更新的最坏复杂度为O（N^2）。

要注意的是，尽管连锁更新的复杂度较高，但它真正造成性能问题

的几率是很低的：

·首先，压缩列表里要恰好有多个连续的、长度介于250字节至253字节之间的节点，连锁更新才有可能被引发，在实际中，这种情况并不多见；

·其次，即使出现连锁更新，但只要被更新的节点数量不多，就不会对性能造成任何影响：比如说，对三五个节点进行连锁更新是绝对不会影响性能的；

因为以上原因，ziplistPush等命令的平均复杂度仅为O（N），在实际中，我们可以放心地使用这些函数，而不必担心连锁更新会影响压缩列表的性能。

7.4　压缩列表API

表7-4列出了所有用于操作压缩列表的API。

表7-4　压缩列表API

函数	作用	算法复杂度
ziplistNew	创建一个新的压缩列表	$O(1)$
ziplistPush	创建一个包含给定值的新节点，并将这个新节点添加到压缩列表的表头或者表尾	平均 $O(N)$，最坏 $O(N^2)$
ziplistInsert	将包含给定值的新节点插入到给定节点之后	平均 $O(N)$，最坏 $O(N^2)$
ziplistIndex	返回压缩列表给定索引上的节点	$O(N)$
ziplistFind	在压缩列表中查找并返回包含了给定值的节点	因为节点的值可能是一个字节数组，所以检查节点值和给定值是否相同的复杂度为 $O(N)$，而查找整个列表的复杂度则为 $O(N^2)$
ziplistNext	返回给定节点的下一个节点	$O(1)$
ziplistPrev	返回给定节点的前一个节点	$O(1)$
ziplistGet	获取给定节点所保存的值	$O(1)$
ziplistDelete	从压缩列表中删除给定的节点	平均 $O(N)$，最坏 $O(N^2)$
ziplistDeleteRange	删除压缩列表在给定索引上的连续多个节点	平均 $O(N)$，最坏 $O(N^2)$
ziplistBlobLen	返回压缩列表目前占用的内存字节数	$O(1)$
ziplistLen	返回压缩列表目前包含的节点数量	节点数量小于 65 535 时为 $O(1)$，大于 65 535 时为 $O(N)$

因为ziplistPush、ziplistInsert、ziplistDelete和ziplistDeleteRange四个函数都有可能会引发连锁更新，所以它们的最坏复杂度都是O（N^2）。

7.5 重点回顾

·压缩列表是一种为节约内存而开发的顺序型数据结构。

·压缩列表被用作列表键和哈希键的底层实现之一。

·压缩列表可以包含多个节点，每个节点可以保存一个字节数组或者整数值。

·添加新节点到压缩列表，或者从压缩列表中删除节点，可能会引发连锁更新操作，但这种操作出现的几率并不高。

第8章　对象

在前面的数个章节里，我们陆续介绍了Redis用到的所有主要数据结构，比如简单动态字符串（SDS）、双端链表、字典、压缩列表、整数集合等等。

Redis并没有直接使用这些数据结构来实现键值对数据库，而是基于这些数据结构创建了一个对象系统，这个系统包含字符串对象、列表对象、哈希对象、集合对象和有序集合对象这五种类型的对象，每种对象都用到了至少一种我们前面所介绍的数据结构。

通过这五种不同类型的对象，Redis可以在执行命令之前，根据对象的类型来判断一个对象是否可以执行给定的命令。使用对象的另一个好处是，我们可以针对不同的使用场景，为对象设置多种不同的数据结构实现，从而优化对象在不同场景下的使用效率。

除此之外，Redis的对象系统还实现了基于引用计数技术的内存回收机制，当程序不再使用某个对象的时候，这个对象所占用的内存就会被自动释放；另外，Redis还通过引用计数技术实现了对象共享机制，这一机制可以在适当的条件下，通过让多个数据库键共享同一个对象来节约内存。

最后，Redis的对象带有访问时间记录信息，该信息可以用于计算数据库键的空转时长，在服务器启用了maxmemory功能的情况下，空转时长较大的那些键可能会优先被服务器删除。

本章接下来将逐一介绍以上提到的Redis对象系统的各个特性。

8.1　对象的类型与编码

Redis使用对象来表示数据库中的键和值，每次当我们在Redis的数据库中新创建一个键值对时，我们至少会创建两个对象，一个对象用作键值对的键（键对象），另一个对象用作键值对的值（值对象）。

举个例子，以下SET命令在数据库中创建了一个新的键值对，其中键值对的键是一个包含了字符串值"msg"的对象，而键值对的值则是一个包含了字符串值"hello world"的对象：

```
redis> SET msg "hello world"
OK
```

Redis中的每个对象都由一个redisObject结构表示，该结构中和保存数据有关的三个属性分别是type属性、encoding属性和ptr属性：

```
typedef struct redisObject {
    //
类型
    unsigned type:4;
    //
编码
    unsigned encoding:4;
    //
指向底层实现数据结构的指针
    void *ptr;
    // ...
} robj;
```

8.1.1　类型

对象的type属性记录了对象的类型，这个属性的值可以是表8-1列出的常量的其中一个。

表8-1　对象的类型

类型常量	对象的名称
REDIS_STRING	字符串对象
REDIS_LIST	列表对象
REDIS_HASH	哈希对象
REDIS_SET	集合对象
REDIS_ZSET	有序集合对象

对于Redis数据库保存的键值对来说，键总是一个字符串对象，而值则可以是字符串对象、列表对象、哈希对象、集合对象或者有序集合对象的其中一种，因此：

·当我们称呼一个数据库键为“字符串键”时，我们指的是“这个数据库键所对应的值为字符串对象”；

·当我们称呼一个键为“列表键”时，我们指的是“这个数据库键所对应的值为列表对象”。

TYPE命令的实现方式也与此类似，当我们对一个数据库键执行TYPE命令时，命令返回的结果为数据库键对应的值对象的类型，而不是键对象的类型：

```
#
键为字符串对象，值为字符串对象
redis> SET msg "hello world"
OK
redis> TYPE msg
string
#
键为字符串对象，值为列表对象
redis> RPUSH numbers 1 3 5
(integer) 6
redis> TYPE numbers
list
#
键为字符串对象，值为哈希对象
redis> HMSET profile name Tom age 25 career Programmer
OK
redis> TYPE profile
hash
#
键为字符串对象，值为集合对象
redis> SADD fruits apple banana cherry
```

```
(integer) 3
redis> TYPE fruits
set
#
键为字符串对象，值为有序集合对象
redis> ZADD price 8.5 apple 5.0 banana 6.0 cherry
(integer) 3
redis> TYPE price
zset
```

表8-2列出了TYPE命令在面对不同类型的值对象时所产生的输出。

表8-2　不同类型值对象的TYPE命令输出

对象	对象 type 属性的值	*TYPE* 命令的输出
字符串对象	REDIS_STRING	"string"
列表对象	REDIS_LIST	"list"
哈希对象	REDIS_HASH	"hash"
集合对象	REDIS_SET	"set"
有序集合对象	REDIS_ZSET	"zset"

8.1.2　编码和底层实现

对象的ptr指针指向对象的底层实现数据结构，而这些数据结构由对象的encoding属性决定。

encoding属性记录了对象所使用的编码，也即是说这个对象使用了什么数据结构作为对象的底层实现，这个属性的值可以是表8-3列出的常量的其中一个。

表8-3　对象的编码

编码常量	编码所对应的底层数据结构
REDIS_ENCODING_INT	long 类型的整数
REDIS_ENCODING_EMBSTR	embstr 编码的简单动态字符串
REDIS_ENCODING_RAW	简单动态字符串
REDIS_ENCODING_HT	字典
REDIS_ENCODING_LINKEDLIST	双端链表
REDIS_ENCODING_ZIPLIST	压缩列表
REDIS_ENCODING_INTSET	整数集合
REDIS_ENCODING_SKIPLIST	跳跃表和字典

每种类型的对象都至少使用了两种不同的编码，表8-4列出了每种类型的对象可以使用的编码。

表8-4　不同类型和编码的对象

类 型	编 码	对 象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用 embstr 编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象

使用OBJECT ENCODING命令可以查看一个数据库键的值对象的编码：

```
redis> SET msg "hello wrold"
OK
redis> OBJECT ENCODING msg
"embstr"
redis> SET story "long long long long long long ago ..."
OK
redis> OBJECT ENCODING story
"raw"
redis> SADD numbers 1 3 5
(integer) 3
redis> OBJECT ENCODING numbers
"intset"
redis> SADD numbers "seven"
(integer) 1
redis> OBJECT ENCODING numbers
"hashtable"
```

表8-5列出了不同编码的对象所对应的OBJECT ENCODING命令输出。

表8-5 OBJECT ENCODING对不同编码的输出

对象所使用的底层数据结构	编码常量	OBJECT ENCODING 命令输出
整数	REDIS_ENCODING_INT	"int"
embstr 编码的简单动态字符串（SDS）	REDIS_ENCODING_EMBSTR	"embstr"
简单动态字符串	REDIS_ENCODING_RAW	"raw"
字典	REDIS_ENCODING_HT	"hashtable"

（续）

对象所使用的底层数据结构	编码常量	OBJECT ENCODING 命令输出
双端链表	REDIS_ENCODING_LINKEDLIST	"linkedlist"
压缩列表	REDIS_ENCODING_ZIPLIST	"ziplist"
整数集合	REDIS_ENCODING_INTSET	"intset"
跳跃表和字典	REDIS_ENCODING_SKIPLIST	"skiplist"

通过encoding属性来设定对象所使用的编码，而不是为特定类型的对象关联一种固定的编码，极大地提升了Redis的灵活性和效率，因为

Redis可以根据不同的使用场景来为一个对象设置不同的编码，从而优化对象在某一场景下的效率。

举个例子，在列表对象包含的元素比较少时，Redis使用压缩列表作为列表对象的底层实现：

·因为压缩列表比双端链表更节约内存，并且在元素数量较少时，在内存中以连续块方式保存的压缩列表比起双端链表可以更快被载入到缓存中；

·随着列表对象包含的元素越来越多，使用压缩列表来保存元素的优势逐渐消失时，对象就会将底层实现从压缩列表转向功能更强、也更适合保存大量元素的双端链表上面；

其他类型的对象也会通过使用多种不同的编码来进行类似的优化。

在接下来的内容中，我们将分别介绍Redis中的五种不同类型的对象，说明这些对象底层所使用的编码方式，列出对象从一种编码转换成另一种编码所需的条件，以及同一个命令在多种不同编码上的实现方法。

8.2 字符串对象

字符串对象的编码可以是int、raw或者embstr。

如果一个字符串对象保存的是整数值，并且这个整数值可以用long类型来表示，那么字符串对象会将整数值保存在字符串对象结构的ptr属性里面（将void*转换成long），并将字符串对象的编码设置为int。

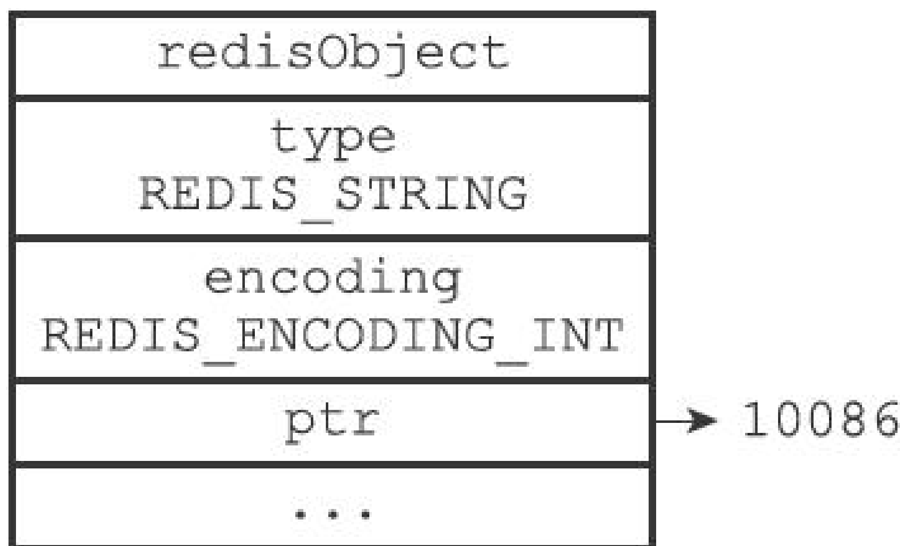


图8-1　int编码的字符串对象

举个例子，如果我们执行以下SET命令，那么服务器将创建一个如图8-1所示的int编码的字符串对象作为number键的值：

```
redis> SET number 10086
OK
redis> OBJECT ENCODING number
"int"
```

如果字符串对象保存的是一个字符串值，并且这个字符串值的长度大于32字节，那么字符串对象将使用一个简单动态字符串（SDS）来保存这个字符串值，并将对象的编码设置为raw。

举个例子，如果我们执行以下命令，那么服务器将创建一个如图8-2所示的raw编码的字符串对象作为story键的值：

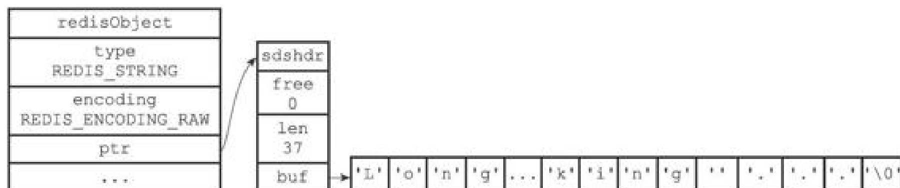


图8-2　raw编码的字符串对象

```
redis> SET story "Long, long ago there lived a king ..."
OK
redis> STRLEN story
(integer) 37
redis> OBJECT ENCODING story
"raw"
```

如果字符串对象保存的是一个字符串值，并且这个字符串值的长度小于等于32字节，那么字符串对象将使用embstr编码的方式来保存这个字符串值。

embstr编码是专门用于保存短字符串的一种优化编码方式，这种编码和raw编码一样，都使用redisObject结构和sdshdr结构来表示字符串对象，但raw编码会调用两次内存分配函数来分别创建redisObject结构和sdshdr结构，而embstr编码则通过调用一次内存分配函数来分配一块连续的空间，空间中依次包含redisObject和sdshdr两个结构，如图8-3所示。



图8-3　embstr编码创建的内存块结构

embstr编码的字符串对象在执行命令时，产生的效果和raw编码的字符串对象执行命令时产生的效果是相同的，但使用embstr编码的字符串对象来保存短字符串值有以下好处：

·embstr编码将创建字符串对象所需的内存分配次数从raw编码的两次降低为一次。

·释放embstr编码的字符串对象只需要调用一次内存释放函数，而释放raw编码的字符串对象需要调用两次内存释放函数。

·因为embstr编码的字符串对象的所有数据都保存在一块连续的内存里面，所以这种编码的字符串对象比起raw编码的字符串对象能够更好地利用缓存带来的优势。

作为例子，以下命令创建了一个embstr编码的字符串对象作为msg键的值，值对象的样子如图8-4所示：

```
redis> SET msg "hello"
OK
redis> OBJECT ENCODING msg
"embstr"
```



图8-4　embstr编码的字符串对象

最后要说的是，可以用long　double类型表示的浮点数在Redis中也是作为字符串值来保存的。如果我们要保存一个浮点数到字符串对象里面，那么程序会先将这个浮点数转换成字符串值，然后再保存转换所得的字符串值。

举个例子，执行以下代码将创建一个包含3.14的字符串表示“3.14”的字符串对象：

```
redis> SET pi 3.14
OK
redis> OBJECT ENCODING pi
"embstr"
```

在有需要的时候，程序会将保存在字符串对象里面的字符串值转换回浮点数值，执行某些操作，然后再将执行操作所得的浮点数值转换回字符串值，并继续保存在字符串对象里面。

举个例子，如果我们执行以下代码：

```
redis> INCRBYFLOAT pi 2.0
"5.14"
redis> OBJECT ENCODING pi
"embstr"
```

那么程序首先会取出字符串对象里面保存的字符串值"3.14"，将它转换回浮点数值3.14，然后把3.14和2.0相加得出的值5.14转换成字符串"5.14"，并将这个"5.14"保存到字符串对象里面。表8-6总结并列出了字符串对象保存各种不同类型的值所使用的编码方式。

表8-6　字符串对象保存各类型值的编码方式

值	编码
可以用 long 类型保存的整数	int
可以用 long double 类型保存的浮点数	embstr 或者 raw
字符串值，或者因为长度太大而没办法用 long 类型表示的整数，又或者因为长度太大而没办法用 long double 类型表示的浮点数	embstr 或者 raw

8.2.1　编码的转换

int编码的字符串对象和embstr编码的字符串对象在条件满足的情况下，会被转换为raw编码的字符串对象。

对于int编码的字符串对象来说，如果我们向对象执行了一些命令，使得这个对象保存的不再是整数值，而是一个字符串值，那么字符串对象的编码将从int变为raw。

在下面的示例中，我们通过APPEND命令，向一个保存整数值的字符串对象追加了一个字符串值，因为追加操作只能对字符串值执行，所以程序会先将之前保存的整数值10086转换为字符串值"10086"，然后再执行追加操作，操作的执行结果就是一个raw编码的、保存了字符串值的字符串对象：

```
redis> SET number 10086
OK
redis> OBJECT ENCODING number
"int"
redis> APPEND number " is a good number!"
(integer) 23
redis> GET number
"10086 is a good number!"
redis> OBJECT ENCODING number
"raw"
```

另外，因为Redis没有为embstr编码的字符串对象编写任何相应的修改程序（只有int编码的字符串对象和raw编码的字符串对象有这些程序），所以embstr编码的字符串对象实际上是只读的。当我们对embstr编码的字符串对象执行任何修改命令时，程序会先将对象的编码从embstr转换成raw，然后再执行修改命令。因为这个原因，embstr编码的字符串对象在执行修改命令之后，总会变成一个raw编码的字符串对象。

以下代码展示了一个embstr编码的字符串对象在执行APPEND命令之后，对象的编码从embstr变为raw的例子：

```
redis> SET msg "hello world"
OK
redis> OBJECT ENCODING msg
"embstr"
redis> APPEND msg " again!"
(integer) 18
redis> OBJECT ENCODING msg
"raw"
```

8.2.2 字符串命令的实现

因为字符串键的值为字符串对象，所以用于字符串键的所有命令都是针对字符串对象来构建的，表8-7列举了其中一部分字符串命令，以及这些命令在不同编码的字符串对象下的实现方法。

表8-7 字符串命令的实现

命令	int 编码的实现方法	embstr 编码的实现方法	raw 编码的实现方法
SET	使用 `int` 编码保存值	使用 `embstr` 编码保存值	使用 `raw` 编码保存值
GET	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后向客户端返回这个字符串值	直接向客户端返回字符串值	直接向客户端返回字符串值
APPEND	将对象转换成 `raw` 编码，然后按 `raw` 编码的方式执行此操作	将对象转换成 `raw` 编码，然后按 `raw` 编码的方式执行此操作	调用 `sdscatlen` 函数，将给定字符串追加到现有字符串的末尾
INCRBYFLOAT	取出整数值并将其转换成 `long double` 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来	取出字符串值并尝试将其转换成 `long double` 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误	取出字符串值并尝试将其转换成 `long double` 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误
INCRBY	对整数值进行加法计算，得出的计算结果会作为整数被保存起来	`embstr` 编码不能执行此命令，向客户端返回一个错误	`raw` 编码不能执行此命令，向客户端返回一个错误
DECRBY	对整数值进行减法计算，得出的计算结果会作为整数被保存起来	`embstr` 编码不能执行此命令，向客户端返回一个错误	`raw` 编码不能执行此命令，向客户端返回一个错误
STRLEN	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，计算并返回这个字符串值的长度	调用 `sdslen` 函数，返回字符串的长度	调用 `sdslen` 函数，返回字符串的长度
SETRANGE	将对象转换成 `raw` 编码，然后按 `raw` 编码的方式执行此命令	将对象转换成 `raw` 编码，然后按 `raw` 编码的方式执行此命令	将字符串特定索引上的值设置为给定的字符
GETRANGE	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符

8.3　列表对象

列表对象的编码可以是ziplist或者linkedlist。

ziplist编码的列表对象使用压缩列表作为底层实现，每个压缩列表节点（entry）保存了一个列表元素。举个例子，如果我们执行以下RPUSH命令，那么服务器将创建一个列表对象作为numbers键的值：

```
redis> RPUSH numbers 1 "three" 5
(integer) 3
```

如果numbers键的值对象使用的是ziplist编码，这个这个值对象将会是图8-5所展示的样子。

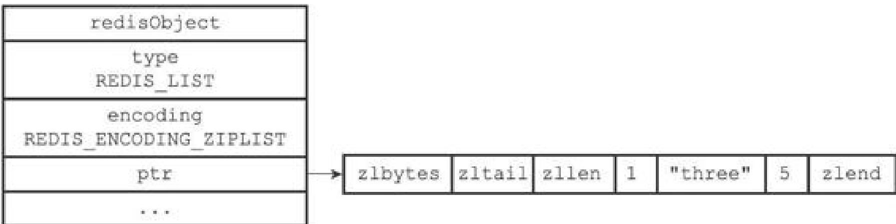


图8-5　ziplist编码的numbers列表对象

另一方面，linkedlist编码的列表对象使用双端链表作为底层实现，每个双端链表节点（node）都保存了一个字符串对象，而每个字符串对象都保存了一个列表元素。

举个例子，如果前面所说的numbers键创建的列表对象使用的不是ziplist编码，而是linkedlist编码，那么numbers键的值对象将是图8-6所示的样子。

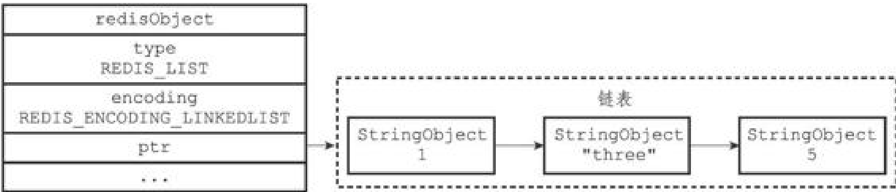


图8-6　linkedlist编码的numbers列表对象

注意，linkedlist编码的列表对象在底层的双端链表结构中包含了多个字符串对象，这种嵌套字符串对象的行为在稍后介绍的哈希对象、集合对象和有序集合对象中都会出现，字符串对象是Redis五种类型的对象中唯一一种会被其他四种类型对象嵌套的对象。


注意

为了简化字符串对象的表示，我们在图8-6使用了一个带有StringObject字样的格子来表示一个字符串对象，而StringObject字样下面的是字符串对象所保存的值。比如说，图8-7代表的就是一个包含了字符串值"three"的字符串对象，它是图8-8的简化表示。

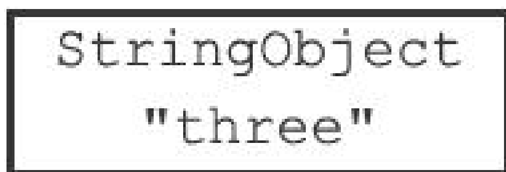


图8-7　简化的字符串对象表示



图8-8　完整的字符串对象表示

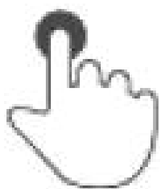
本书接下来的内容将继续沿用这一简化表示。

8.3.1　编码转换

当列表对象可以同时满足以下两个条件时，列表对象使用ziplist编码：

·列表对象保存的所有字符串元素的长度都小于64字节；

·列表对象保存的元素数量小于512个；不能满足这两个条件的列表对象需要使用linkedlist编码。


注意

以上两个条件的上限值是可以修改的，具体请看配置文件中关于list-max-ziplist-value选项和list-max-ziplist-entries选项的说明。

对于使用ziplist编码的列表对象来说，当使用ziplist编码所需的两个

条件的任意一个不能被满足时，对象的编码转换操作就会被执行，原本保存在压缩列表里的所有列表元素都会被转移并保存到双端链表里面，对象的编码也会从ziplist变为linkedlist。

以下代码展示了列表对象因为保存了长度太大的元素而进行编码转换的情况：

```
#
所有元素的长度都小于64
字节
redis> RPUSH blah "hello" "world" "again"
(integer)3
redis> OBJECT ENCODING blah
"ziplist"
#
将一个65
字节长的元素推入列表对象中
redis> RPUSH blah "wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww"
(integer) 4
#
编码已改变
redis> OBJECT ENCODING blah
"linkedlist"
```

除此之外，以下代码展示了列表对象因为保存的元素数量过多而进行编码转换的情况：

```
#
列表对象包含512
个元素
redis> EVAL "for i=1, 512 do redis.call('RPUSH', KEYS[1],i)end" 1 "integers"
(nil)
redis> LLEN integers
(integer) 512
redis> OBJECT ENCODING integers
"ziplist"
#
再向列表对象推入一个新元素，使得对象保存的元素数量达到513
个
redis> RPUSH integers 513
(integer) 513
#
编码已改变
redis> OBJECT ENCODING integers
"linkedlist"
```

8.3.2　列表命令的实现

因为列表键的值为列表对象，所以用于列表键的所有命令都是针对列表对象来构建的，表8-8列出了其中一部分列表键命令，以及这些命令在不同编码的列表对象下的实现方法。

表8-8　列表命令的实现

命令	ziplist 编码的实现方法	linkedlist 编码的实现方法
LPUSH	调用 `ziplistPush` 函数，将新元素推入到压缩列表的表头	调用 `listAddNodeHead` 函数，将新元素推入到双端链表的表头
RPUSH	调用 `ziplistPush` 函数，将新元素推入到压缩列表的表尾	调用 `listAddNodeTail` 函数，将新元素推入到双端链表的表尾
LPOP	调用 `ziplistIndex` 函数定位压缩列表的表头节点，在向用户返回节点所保存的元素之后，调用 `ziplistDelete` 函数删除表头节点	调用 `listFirst` 函数定位双端链表的表头节点，在向用户返回节点所保存的元素之后，调用 `listDelNode` 函数删除表头节点
RPOP	调用 `ziplistIndex` 函数定位压缩列表的表尾节点，在向用户返回节点所保存的元素之后，调用 `ziplistDelete` 函数删除表尾节点	调用 `listLast` 函数定位双端链表的表尾节点，在向用户返回节点所保存的元素之后，调用 `listDelNode` 函数删除表尾节点
LINDEX	调用 `ziplistIndex` 函数定位压缩列表中的指定节点，然后返回节点所保存的元素	调用 `listIndex` 函数定位双端链表中的指定节点，然后返回节点所保存的元素
LLEN	调用 `ziplistLen` 函数返回压缩列表的长度	调用 `listLength` 函数返回双端链表的长度
LINSERT	插入新节点到压缩列表的表头或者表尾时，使用 `ziplistPush` 函数；插入新节点到压缩列表的其他位置时，使用 `ziplistInsert` 函数	调用 `listInsertNode` 函数，将新节点插入到双端链表的指定位置
LREM	遍历压缩列表节点，并调用 `ziplistDelete` 函数删除包含了给定元素的节点	遍历双端链表节点，并调用 `listDelNode` 函数删除包含了给定元素的节点
LTRIM	调用 `ziplistDeleteRange` 函数，删除压缩列表中所有不在指定索引范围内的节点	遍历双端链表节点，并调用 `listDelNode` 函数删除链表中所有不在指定索引范围内的节点
LSET	调用 `ziplistDelete` 函数，先删除压缩列表指定索引上的现有节点，然后调用 `ziplistInsert` 函数，将一个包含给定元素的新节点插入到相同索引上面	调用 `listIndex` 函数，定位到双端链表指定索引上的节点，然后通过赋值操作更新节点的值

8.4　哈希对象

哈希对象的编码可以是ziplist或者hashtable。

ziplist编码的哈希对象使用压缩列表作为底层实现，每当有新的键值对要加入到哈希对象时，程序会先将保存了键的压缩列表节点推入到压缩列表表尾，然后再将保存了值的压缩列表节点推入到压缩列表表尾，因此：

·保存了同一键值对的两个节点总是紧挨在一起，保存键的节点在前，保存值的节点在后；

·先添加到哈希对象中的键值对会被放在压缩列表的表头方向，而后来添加到哈希对象中的键值对会被放在压缩列表的表尾方向。

举个例子，如果我们执行以下HSET命令，那么服务器将创建一个列表对象作为profile键的值：

```
redis> HSET profile name "Tom"
(integer) 1
redis> HSET profile age 25
(integer) 1
redis> HSET profile career "Programmer"
(integer) 1
```

如果profile键的值对象使用的是ziplist编码，那么这个值对象将会是图8-9所示的样子，其中对象所使用的压缩列表如图8-10所示。

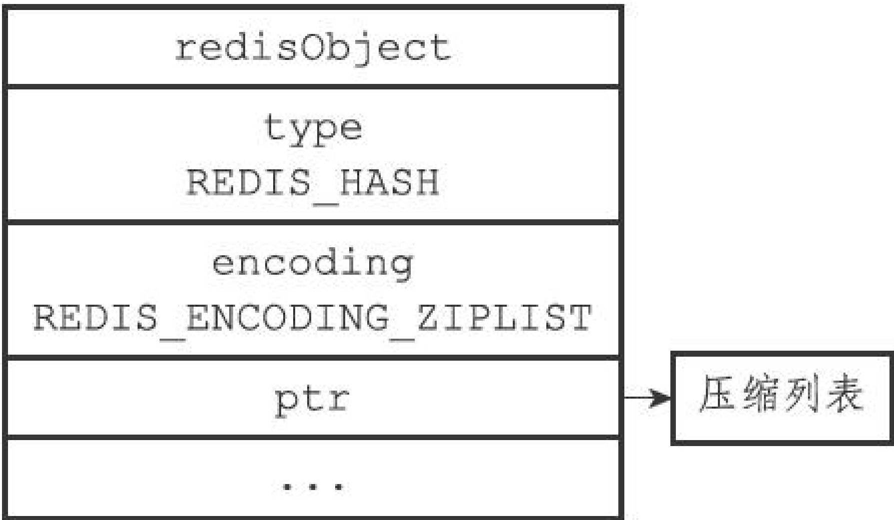


图8-9　ziplist编码的profile哈希对象



图8-10　profile哈希对象的压缩列表底层实现

另一方面，hashtable编码的哈希对象使用字典作为底层实现，哈希对象中的每个键值对都使用一个字典键值对来保存：

·字典的每个键都是一个字符串对象，对象中保存了键值对的键；

·字典的每个值都是一个字符串对象，对象中保存了键值对的值。

举个例子，如果前面profile键创建的不是ziplist编码的哈希对象，而是hashtable编码的哈希对象，那么这个哈希对象应该会是图8-11所示的样子。

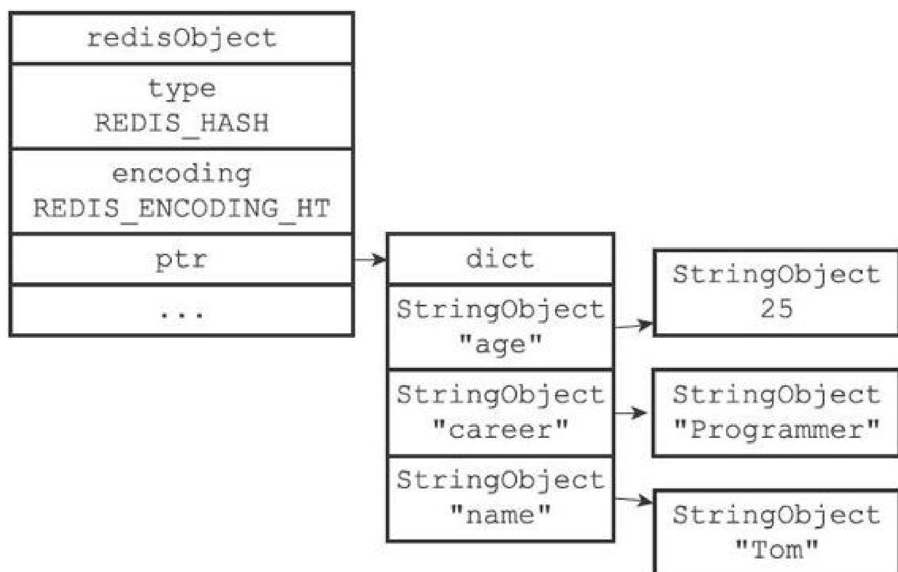


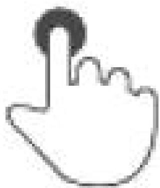
图8-11　hashtable编码的profile哈希对象

8.4.1　编码转换

当哈希对象可以同时满足以下两个条件时，哈希对象使用ziplist编码：

·哈希对象保存的所有键值对的键和值的字符串长度都小于64字节；

·哈希对象保存的键值对数量小于512个；不能满足这两个条件的哈希对象需要使用hashtable编码。

注意

这两个条件的上限值是可以修改的，具体请看配置文件中关于hash-max-ziplist-value选项和hash-max-ziplist-entries选项的说明。

对于使用ziplist编码的列表对象来说，当使用ziplist编码所需的两个条件的任意一个不能被满足时，对象的编码转换操作就会被执行，原本保存在压缩列表里的所有键值对都会被转移并保存到字典里面，对象的编码也会从ziplist变为hashtable。

以下代码展示了哈希对象因为键值对的键长度太大而引起编码转换的情况：

```
#
哈希对象只包含一个键和值都不超过64
个字节的键值对
redis> HSET book name "Mastering C++ in 21 days"
(integer) 1
redis> OBJECT ENCODING book
"ziplist"
#
向哈希对象添加一个新的键值对，键的长度为66
字节
redis> HSET book long_long_long_long_long_long_long_long_long_long_long_description "
(integer) 1
#
编码已改变
redis> OBJECT ENCODING book
"hashtable"
```

除了键的长度太大会引起编码转换之外，值的长度太大也会引起编码转换，以下代码展示了这种情况的一个示例：

```
#
哈希对象只包含一个键和值都不超过64
个字节的键值对
redis> HSET blah greeting "hello world"
(integer) 1
redis> OBJECT ENCODING blah
"ziplist"
#
向哈希对象添加一个新的键值对，值的长度为68
字节
redis> HSET blah story "many string ... many string ... many string ... many string .
(integer) 1
#
编码已改变
redis> OBJECT ENCODING blah
"hashtable"
```

最后，以下代码展示了哈希对象因为包含的键值对数量过多而引起编码转换的情况：

```
#
创建一个包含512
个键值对的哈希对象
redis> EVAL "for i=1, 512 do redis.call('HSET', KEYS[1], i, i)end" 1 "numbers"
(nil)
redis> HLEN numbers
(integer) 512
redis> OBJECT ENCODING numbers
"ziplist"
#
再向哈希对象添加一个新的键值对，使得键值对的数量变成513
个
redis> HMSET numbers "key" "value"
OK
redis> HLEN numbers
(integer) 513
#
编码改变
redis> OBJECT ENCODING numbers
"hashtable"
```

8.4.2 哈希命令的实现

因为哈希键的值为哈希对象，所以用于哈希键的所有命令都是针对哈希对象来构建的，表8-9列出了其中一部分哈希键命令，以及这些命令在不同编码的哈希对象下的实现方法。

表8-9 哈希命令的实现

命令	ziplist 编码实现方法	hashtable 编码的实现方法
HSET	首先调用 ziplistPush 函数，将键推入到压缩列表的表尾，然后再次调用 ziplistPush 函数，将值推入到压缩列表的表尾	调用 dictAdd 函数，将新节点添加到字典里面
HGET	首先调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后调用 ziplistNext 函数，将指针移动到键节点旁边的值节点，最后返回值节点	调用 dictFind 函数，在字典中查找给定键，然后调用 dictGetVal 函数，返回该键所对应的值
HEXISTS	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，如果找到的话说明键值对存在，没找到的话就说明键值对不存在	调用 dictFind 函数，在字典中查找给定键，如果找到的话说明键值对存在，没找到的话就说明键值对不存在
HDEL	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后将相应的键节点、以及键节点旁边的值节点都删除掉	调用 dictDelete 函数，将指定键所对应的键值对从字典中删除掉
HLEN	调用 ziplistLen 函数，取得压缩列表包含节点的总数量，将这个数量除以 2，得出的结果就是压缩列表保存的键值对的数量	调用 dictSize 函数，返回字典包含的键值对数量，这个数量就是哈希对象包含的键值对数量
HGETALL	遍历整个压缩列表，用 ziplistGet 函数返回所有键和值（都是节点）	遍历整个字典，用 dictGetKey 函数返回字典的键，用 dictGetVal 函数返回字典的值

8.5 集合对象

集合对象的编码可以是intset或者hashtable。

intset编码的集合对象使用整数集合作为底层实现，集合对象包含的所有元素都被保存在整数集合里面。

举个例子，以下代码将创建一个如图8-12所示的intset编码集合对象：

```
redis> SADD numbers 1 3 5
(integer) 3
```

另一方面，hashtable编码的集合对象使用字典作为底层实现，字典的每个键都是一个字符串对象，每个字符串对象包含了一个集合元素，而字典的值则全部被设置为NULL。

举个例子，以下代码将创建一个如图8-13所示的hashtable编码集合对象：

```
redis> SAD Dfruits "apple" "banana" "cherry"
(integer)3
```

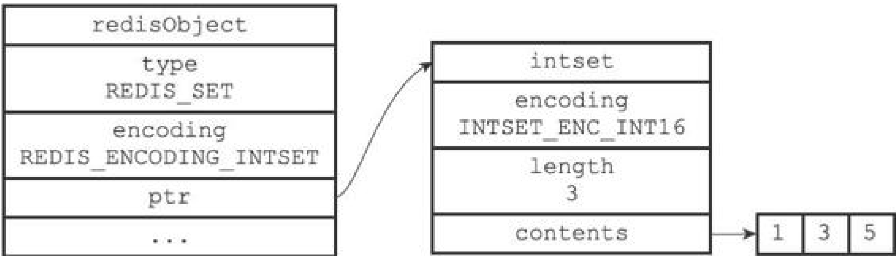


图8-12　intset编码的numbers集合对象

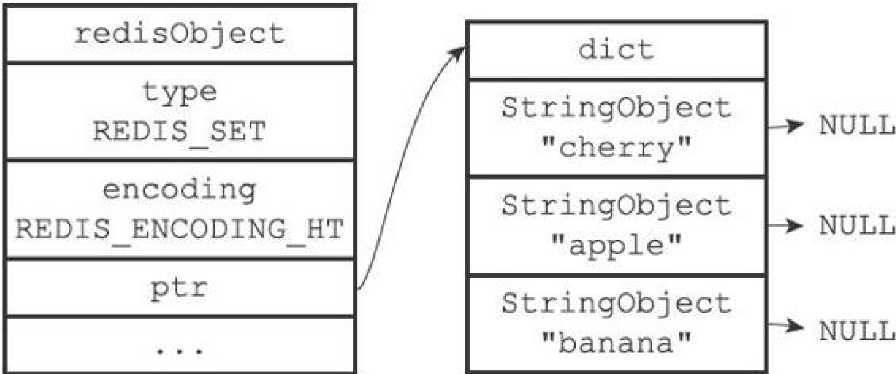


图8-13　hashtable编码的fruits集合对象

8.5.1 编码的转换

当集合对象可以同时满足以下两个条件时，对象使用intset编码：

·集合对象保存的所有元素都是整数值；

·集合对象保存的元素数量不超过512个。

不能满足这两个条件的集合对象需要使用hashtable编码。



注意

第二个条件的上限值是可以修改的，具体请看配置文件中关于set-max-intset-entries选项的说明。

对于使用intset编码的集合对象来说，当使用intset编码所需的两个条件的任意一个不能被满足时，就会执行对象的编码转换操作，原本保存在整数集合中的所有元素都会被转移并保存到字典里面，并且对象的编码也会从intset变为hashtable。

举个例子，以下代码创建了一个只包含整数元素的集合对象，该对象的编码为intset：

```
redis> SADD numbers 1 3 5
(integer) 3
redis> OBJECT ENCODING numbers
"intset"
```

不过，只要我们向这个只包含整数元素的集合对象添加一个字符串元素，集合对象的编码转移操作就会被执行：

```
redis> SADD numbers "seven"
(integer) 1
redis> OBJECT ENCODING numbers
"hashtable"
```

除此之外，如果我们创建一个包含512个整数元素的集合对象，那么对象的编码应该会是intset：

```
redis> EVAL "for i=1, 512 do redis.call('SADD', KEYS[1], i) end" 1 integers
(nil)
redis> SCARD integers
(integer) 512
redis> OBJECT ENCODING integers
"intset"
```

但是，只要我们再向集合添加一个新的整数元素，使得这个集合的元素数量变成513，那么对象的编码转换操作就会被执行：

```
redis> SADD integers 10086
(integer) 1
redis> SCARD integers
(integer) 513
redis> OBJECT ENCODING integers
"hashtable"
```

8.5.2　集合命令的实现

因为集合键的值为集合对象，所以用于集合键的所有命令都是针对集合对象来构建的，表8-10列出了其中一部分集合键命令，以及这些命令在不同编码的集合对象下的实现方法。

表8-10　集合命令的实现方法

命令	intset 编码的实现方法	hashtable 编码的实现方法
SADD	调用 intsetAdd 函数，将所有新元素添加到整数集合里面	调用 dictAdd，以新元素为键，NULL 为值，将键值对添加到字典里面
SCARD	调用 intsetLen 函数，返回整数集合所包含的元素数量，这个数量就是集合对象所包含的元素数量	调用 dictSize 函数，返回字典所包含的键值对数量，这个数量就是集合对象所包含的元素数量
SISMEMBER	调用 intsetFind 函数，在整数集合中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合	调用 dictFind 函数，在字典的键中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合
SMEMBERS	遍历整个整数集合，使用 intsetGet 函数返回集合元素	遍历整个字典，使用 dictGetKey 函数返回字典的键作为集合元素
SRANDMEMBER	调用 intsetRandom 函数，从整数集合中随机返回一个元素	调用 dictGetRandomKey 函数，从字典中随机返回一个字典键
SPOP	调用 intsetRandom 函数，从整数集合中随机取出一个元素，在将这个随机元素返回给客户端之后，调用 intsetRemove 函数，将随机元素从整数集合中删除掉	调用 dictGetRandomKey 函数，从字典中随机取出一个字典键，在将这个随机字典键的值返回给客户端之后，调用 dictDelete 函数，从字典中删除随机字典键所对应的键值对
SREM	调用 intsetRemove 函数，从整数集合中删除所有给定的元素	调用 dictDelete 函数，从字典中删除所有键为给定元素的键值对

（续）

8.6　有序集合对象

有序集合的编码可以是ziplist或者skiplist。

ziplist编码的压缩列表对象使用压缩列表作为底层实现，每个集合元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素的成员（member），而第二个元素则保存元素的分值（score）。

压缩列表内的集合元素按分值从小到大进行排序，分值较小的元素被放置在靠近表头的方向，而分值较大的元素则被放置在靠近表尾的方向。

举个例子，如果我们执行以下ZADD命令，那么服务器将创建一个有序集合对象作为price键的值：

```
redis> ZADD price 8.5 apple 5.0 banana 6.0 cherry
(integer) 3
```

如果price键的值对象使用的是ziplist编码，那么这个值对象将会是图8-14所示的样子，而对象所使用的压缩列表则会是8-15所示的样子。

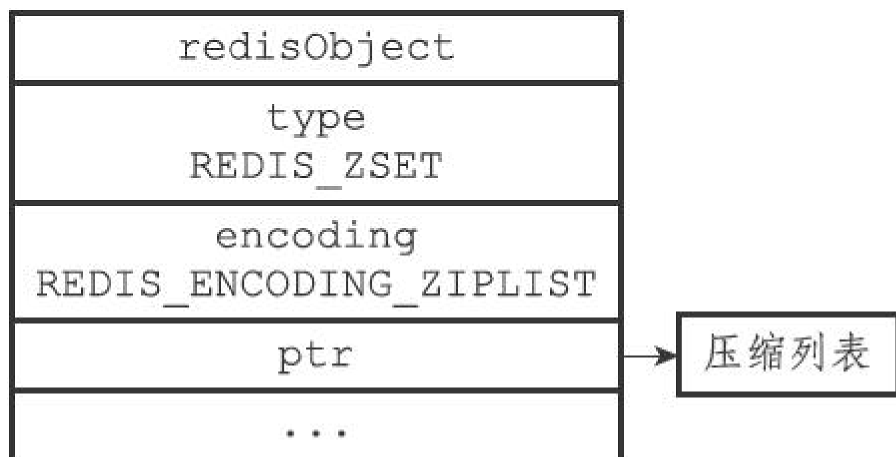


图8-14　ziplist编码的有序集合对象

skiplist编码的有序集合对象使用zset结构作为底层实现，一个zset结构同时包含一个字典和一个跳跃表：

```
typedef struct zset {
    zskiplist *zsl;
    dict *dict;
} zset;
```



图8-15　有序集合元素在压缩列表中按分值从小到大排列

zset结构中的zsl跳跃表按分值从小到大保存了所有集合元素，每个跳跃表节点都保存了一个集合元素：跳跃表节点的object属性保存了元素的成员，而跳跃表节点的score属性则保存了元素的分值。通过这个跳跃表，程序可以对有序集合进行范围型操作，比如ZRANK、ZRANGE等命令就是基于跳跃表API来实现的。

除此之外，zset结构中的dict字典为有序集合创建了一个从成员到分值的映射，字典中的每个键值对都保存了一个集合元素：字典的键保存了元素的成员，而字典的值则保存了元素的分值。通过这个字典，程序可以用O（1）复杂度查找给定成员的分值，ZSCORE命令就是根据这一特性实现的，而很多其他有序集合命令都在实现的内部用到了这一特性。

有序集合每个元素的成员都是一个字符串对象，而每个元素的分值都是一个double类型的浮点数。值得一提的是，虽然zset结构同时使用跳跃表和字典来保存有序集合元素，但这两种数据结构都会通过指针来共享相同元素的成员和分值，所以同时使用跳跃表和字典来保存集合元素不会产生任何重复成员或者分值，也不会因此而浪费额外的内存。

为什么有序集合需要同时使用跳跃表和字典来实现？

在理论上，有序集合可以单独使用字典或者跳跃表的其中一种数据结构来实现，但无论单独使用字典还是跳跃表，在性能上对比起同时使用字典和跳跃表都会有所降低。举个例子，如果我们只使用字典来实现有序集合，那么虽然以O（1）复杂度查找成员的分值这一特性会被保留，但是，因为字典以无序的方式来保存集合元素，所以每次在执行范围型操作——比如ZRANK、ZRANGE等命令时，程序都需要对字典保存的所有元素进行排序，完成这种排序需要至少O（NlogN）时间复杂度，以及额外的O（N）内存空间（因为要创建一个数组来保存排序后的元素）。

另一方面，如果我们只使用跳跃表来实现有序集合，那么跳跃表执行范围型操作的所有优点都会被保留，但因为没有了字典，所以根据成员查找分值这一操作的复杂度将从O（1）上升为O（logN）。因为以上原因，为了让有序集合的查找和范围型操作都尽可能快地执行，Redis选择了同时使用字典和跳跃表两种数据结构来实现有序集合。

举个例子，如果前面price键创建的不是ziplist编码的有序集合对象，而是skiplist编码的有序集合对象，那么这个有序集合对象将会是图8-16所示的样子，而对象所使用的zset结构将会是图8-17所示的样子。

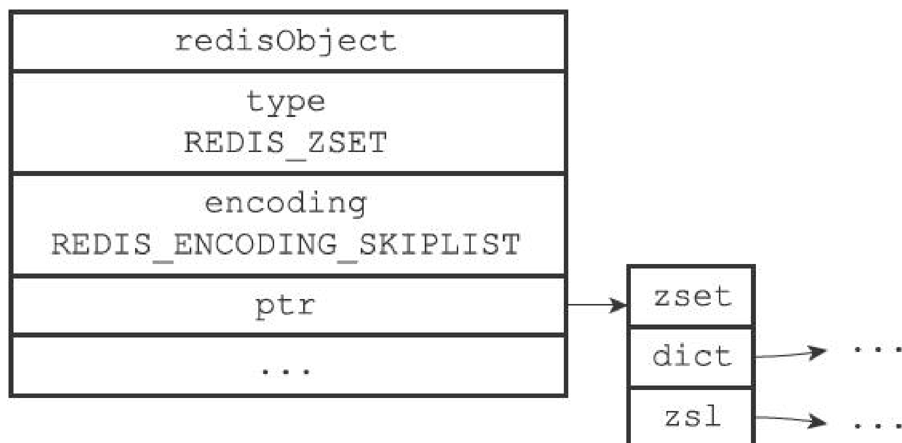


图8-16　skiplist编码的有序集合对象

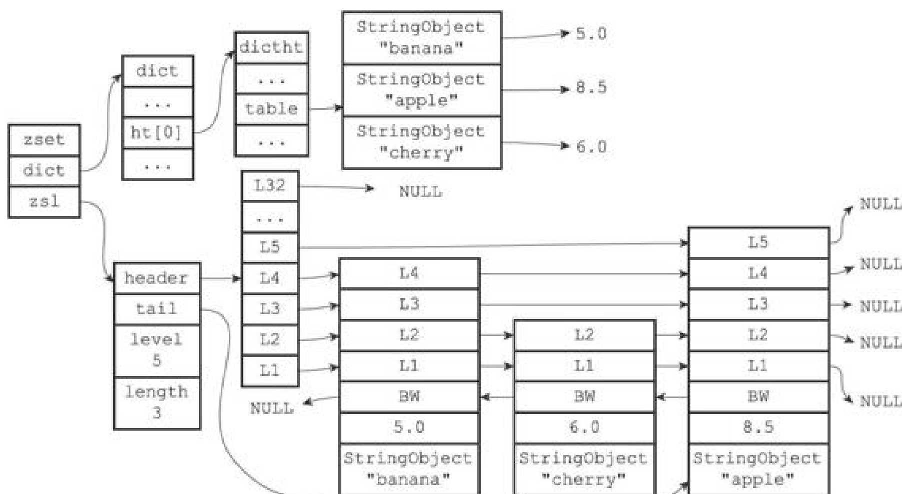


图8-17　有序集合元素同时被保存在字典和跳跃表中



注意

为了展示方便，图8-17在字典和跳跃表中重复展示了各个元素的成员和分值，但在实际中，字典和跳跃表会共享元素的成员和分值，所以并不会造成任何数据重复，也不会因此而浪费任何内存。

8.6.1　编码的转换

当有序集合对象可以同时满足以下两个条件时，对象使用ziplist编码：

·有序集合保存的元素数量小于128个；

·有序集合保存的所有元素成员的长度都小于64字节；

不能满足以上两个条件的有序集合对象将使用skiplist编码。


注意

以上两个条件的上限值是可以修改的，具体请看配置文件中关于zset-max-ziplist-entries选项和zset-max-ziplist-value选项的说明。

对于使用ziplist编码的有序集合对象来说，当使用ziplist编码所需的两个条件中的任意一个不能被满足时，就会执行对象的编码转换操作，原本保存在压缩列表里的所有集合元素都会被转移并保存到zset结构里面，对象的编码也会从ziplist变为skiplist。

以下代码展示了有序集合对象因为包含了过多元素而引发编码转换的情况：

```
#
对象包含了128
个元素
redis> EVAL "for i=1, 128 do redis.call('ZADD', KEYS[1], i, i) end" 1 numbers
(nil)
redis> ZCARD numbers
(integer) 128
redis> OBJECT ENCODING numbers
"ziplist"
#
再添加一个新元素
redis> ZADD numbers 3.14 pi
(integer) 1
#
对象包含的元素数量变为129
个
redis> ZCARD numbers
(integer) 129
#
编码已改变
redis> OBJECT ENCODING numbers
"skiplist"
```

以下代码则展示了有序集合对象因为元素的成员过长而引发编码转换的情况：

```
#
向有序集合添加一个成员只有三字节长的元素
redis> ZADD blah 1.0 www
(integer) 1
redis> OBJECT ENCODING blah
"ziplist"
#
向有序集合添加一个成员为66
字节长的元素
redis> ZADD blah 2.0 oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
(integer) 1
#
编码已改变
redis> OBJECT ENCODING blah
"skiplist"
```

8.6.2 有序集合命令的实现

因为有序集合键的值为哈希对象，所以用于有序集合键的所有命令都是针对哈希对象来构建的，表8-11列出了其中一部分有序集合键命令，以及这些命令在不同编码的哈希对象下的实现方法。

表8-11 有序集合命令的实现方法

命令	ziplist 编码的实现方法	zset 编码的实现方法
ZADD	调用 ziplistInsert 函数，将成员和分值作为两个节点分别插入到压缩列表	先调用 zslInsert 函数，将新元素添加到跳跃表，然后调用 dictAdd 函数，将新元素关联到字典
ZCARD	调用 ziplistLen 函数，获得压缩列表包含节点的数量，将这个数量除以 2 得出集合元素的数量	访问跳跃表数据结构的 length 属性，直接返回集合元素的数量
ZCOUNT	遍历压缩列表，统计分值在给定范围内的节点的数量	遍历跳跃表，统计分值在给定范围内的节点的数量
ZRANGE	从表头向表尾遍历压缩列表，返回给定索引范围内的所有元素	从表头向表尾遍历跳跃表，返回给定索引范围内的所有元素
ZREVRANGE	从表尾向表头遍历压缩列表，返回给定索引范围内的所有元素	从表尾向表头遍历跳跃表，返回给定索引范围内的所有元素
ZRANK	从表头向表尾遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表头向表尾遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREVRANK	从表尾向表头遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表尾向表头遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREM	遍历压缩列表，删除所有包含给定成员的节点，以及被删除成员节点旁边的分值节点	遍历跳跃表，删除所有包含了给定成员的跳跃表节点。并在字典中解除被删除元素的成员和分值的关联
ZSCORE	遍历压缩列表，查找包含了给定成员的节点，然后取出成员节点旁边的分值节点保存的元素分值	直接从字典中取出给定成员的分值

8.7 类型检查与命令多态

Redis中用于操作键的命令基本上可以分为两种类型。

其中一种命令可以对任何类型的键执行，比如说DEL命令、EXPIRE命令、RENAME命令、TYPE命令、OBJECT命令等。

举个例子，以下代码就展示了使用DEL命令来删除三种不同类型的键：

```
#
字符串键
redis> SET msg "hello"
OK
#
列表键
redis> RPUSH numbers 1 2 3
(integer) 3
#
集合键
redis> SADD fruits apple banana cherry
(integer) 3
redis> DEL msg
(integer) 1
redis> DEL numbers
(integer) 1
redis> DEL fruits
(integer) 1
```

而另一种命令只能对特定类型的键执行，比如说：

·SET、GET、APPEND、STRLEN等命令只能对字符串键执行；

·HDEL、HSET、HGET、HLEN等命令只能对哈希键执行；

·RPUSH、LPOP、LINSERT、LLEN等命令只能对列表键执行；

·SADD、SPOP、SINTER、SCARD等命令只能对集合键执行；

·ZADD、ZCARD、ZRANK、ZSCORE等命令只能对有序集合键执行；

举个例子，我们可以用SET命令创建一个字符串键，然后用GET命令和APPEND命令操作这个键，但如果我们试图对这个字符串键执行只有列表键才能执行的LLEN命令，那么Redis将向我们返回一个类型错误：

```
redis> SET msg "hello world"
OK
redis> GET msg
"hello world"
redis> APPEND msg " again!"
(integer) 18
```

```
redis> GET msg
"hello world again!"
redis> LLEN msg
(error) WRONGTYPE Operation against a key holding the wrong kind of value
```

8.7.1　类型检查的实现

从上面发生类型错误的代码示例可以看出，为了确保只有指定类型的键可以执行某些特定的命令，在执行一个类型特定的命令之前，Redis会先检查输入键的类型是否正确，然后再决定是否执行给定的命令。

类型特定命令所进行的类型检查是通过redisObject结构的type属性来实现的：

·在执行一个类型特定命令之前，服务器会先检查输入数据库键的值对象是否为执行命令所需的类型，如果是的话，服务器就对键执行指定的命令；

·否则，服务器将拒绝执行命令，并向客户端返回一个类型错误。

举个例子，对于LLEN命令来说：

·在执行LLEN命令之前，服务器会先检查输入数据库键的值对象是否为列表类型，也即是，检查值对象redisObject结构type属性的值是否为REDIS_LIST，如果是的话，服务器就对键执行LLEN命令；

·否则的话，服务器就拒绝执行命令并向客户端返回一个类型错误；图8-18展示了这一类型检查过程。

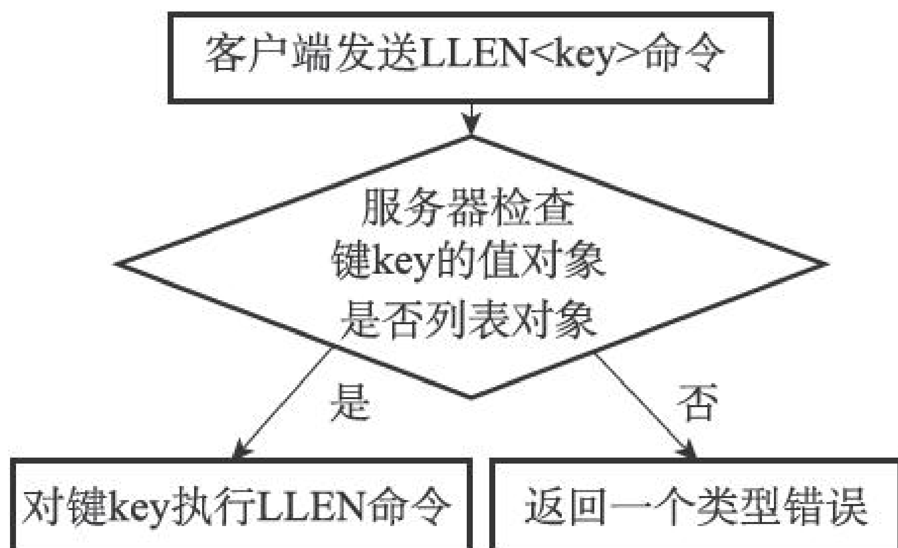


图8-18　LLEN命令执行时的类型检查过程

其他类型特定命令的类型检查过程也和这里展示的LLEN命令的类型检查过程类似。

8.7.2　多态命令的实现

Redis除了会根据值对象的类型来判断键是否能够执行指定命令之外，还会根据值对象的编码方式，选择正确的命令实现代码来执行命令。

举个例子，在前面介绍列表对象的编码时我们说过，列表对象有ziplist和linkedlist两种编码可用，其中前者使用压缩列表API来实现列表命令，而后者则使用双端链表API来实现列表命令。

现在，考虑这样一个情况，如果我们对一个键执行LLEN命令，那么服务器除了要确保执行命令的是列表键之外，还需要根据键的值对象所使用的编码来选择正确的LLEN命令实现：

·如果列表对象的编码为ziplist，那么说明列表对象的实现为压缩列表，程序将使用ziplistLen函数来返回列表的长度；

·如果列表对象的编码为linkedlist，那么说明列表对象的实现为双端链表，程序将使用listLength函数来返回双端链表的长度；

借用面向对象方面的术语来说，我们可以认为LLEN命令是多态（polymorphism）的，只要执行LLEN命令的是列表键，那么无论值对象使用的是ziplist编码还是linkedlist编码，命令都可以正常执行。

图8-19展示了LLEN命令从类型检查到根据编码选择实现函数的整个执行过程，其他类型特定命令的执行过程也是类似的。

实际上，我们可以将DEL、EXPIRE、TYPE等命令也称为多态命令，因为无论输入的键是什么类型，这些命令都可以正确地执行。

DEL、EXPIRE等命令和LLEN等命令的区别在于，前者是基于类型的多态——一个命令可以同时用于处理多种不同类型的键，而后者是基于编码的多态——一个命令可以同时用于处理多种不同编码。

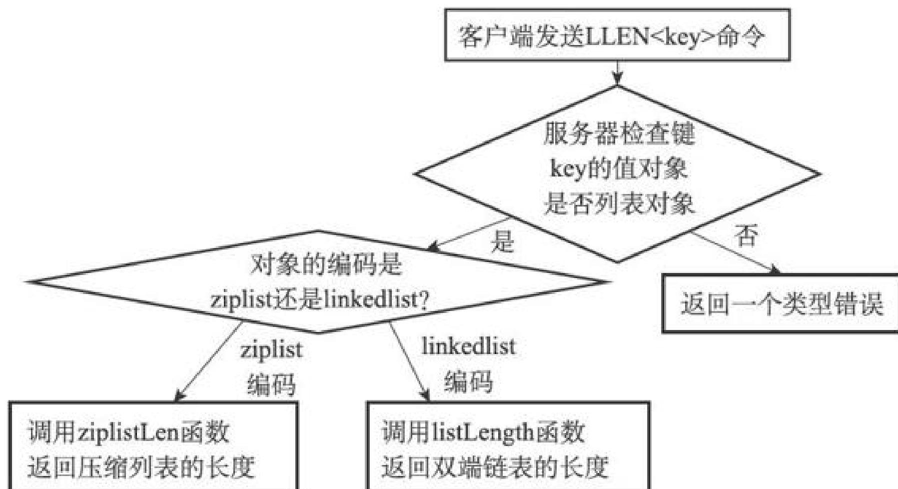


图8-19　LLEN命令的执行过程

8.8 内存回收

因为C语言并不具备自动内存回收功能，所以Redis在自己的对象系统中构建了一个引用计数（reference counting）技术实现的内存回收机制，通过这一机制，程序可以通过跟踪对象的引用计数信息，在适当的时候自动释放对象并进行内存回收。

每个对象的引用计数信息由redisObject结构的refcount属性记录：

```
typedef struct redisObject {
    // ...
    //
引用计数
    int refcount;
    // ...
} robj;
```

对象的引用计数信息会随着对象的使用状态而不断变化：

·在创建一个新对象时，引用计数的值会被初始化为1；

·当对象被一个新程序使用时，它的引用计数值会被增一；

·当对象不再被一个程序使用时，它的引用计数值会被减一；

·当对象的引用计数值变为0时，对象所占用的内存会被释放。

表8-12列出了修改对象引用计数的API，这些API分别用于增加、减少、重置对象的引用计数。

表8-12　修改对象引用计数的API

函数	作用
incrRefCount	将对象的引用计数值增一
decrRefCount	将对象的引用计数值减一，当对象的引用计数值等于0时，释放对象
resetRefCount	将对象的引用计数值设置为0，但并不释放对象，这个函数通常在需要重新设置对象的引用计数值时使用

对象的整个生命周期可以划分为创建对象、操作对象、释放对象三个阶段。作为例子，以下代码展示了一个字符串对象从创建到释放的整个过程：

```
//
创建一个字符串对象s
，对象的引用计数为1
robj *s = createStringObject(...)
//
对象s
执行各种操作...
//
将对象s
的引用计数减一，使得对象的引用计数变为0
```

```
//
导致对象s
被释放
decrRefCount(s)
```

其他不同类型的对象也会经历类似的过程。

8.9 对象共享

除了用于实现引用计数内存回收机制之外，对象的引用计数属性还带有对象共享的作用。举个例子，假设键A创建了一个包含整数值100的字符串对象作为值对象，如图8-20所示。

如果这时键B也要创建一个同样保存了整数值100的字符串对象作为值对象，那么服务器有以下两种做法：

1）为键B新创建一个包含整数值100的字符串对象；

2）让键A和键B共享同一个字符串对象；

以上两种方法很明显是第二种方法更节约内存。

在Redis中，让多个键共享同一个值对象需要执行以下两个步骤：

1）将数据库键的值指针指向一个现有的值对象；

2）将被共享的值对象的引用计数增一。

举个例子，图8-21就展示了包含整数值100的字符串对象同时被键A和键B共享之后的样子，可以看到，除了对象的引用计数从之前的1变成了2之外，其他属性都没有变化。共享对象机制对于节约内存非常有帮助，数据库中保存的相同值对象越多，对象共享机制就能节约越多的内存。

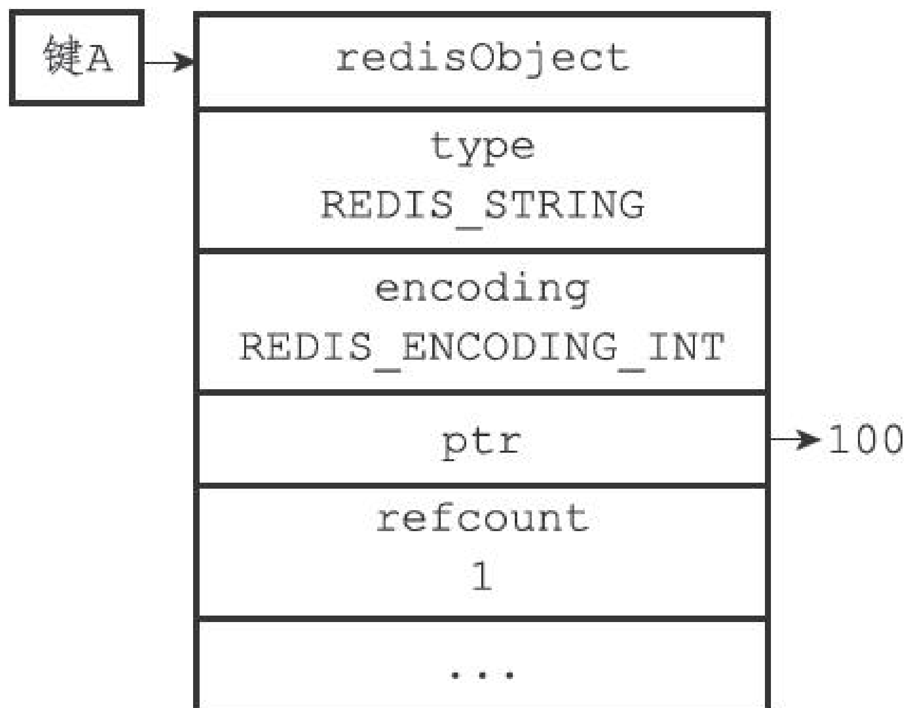


图8-20　未被共享的字符串对象

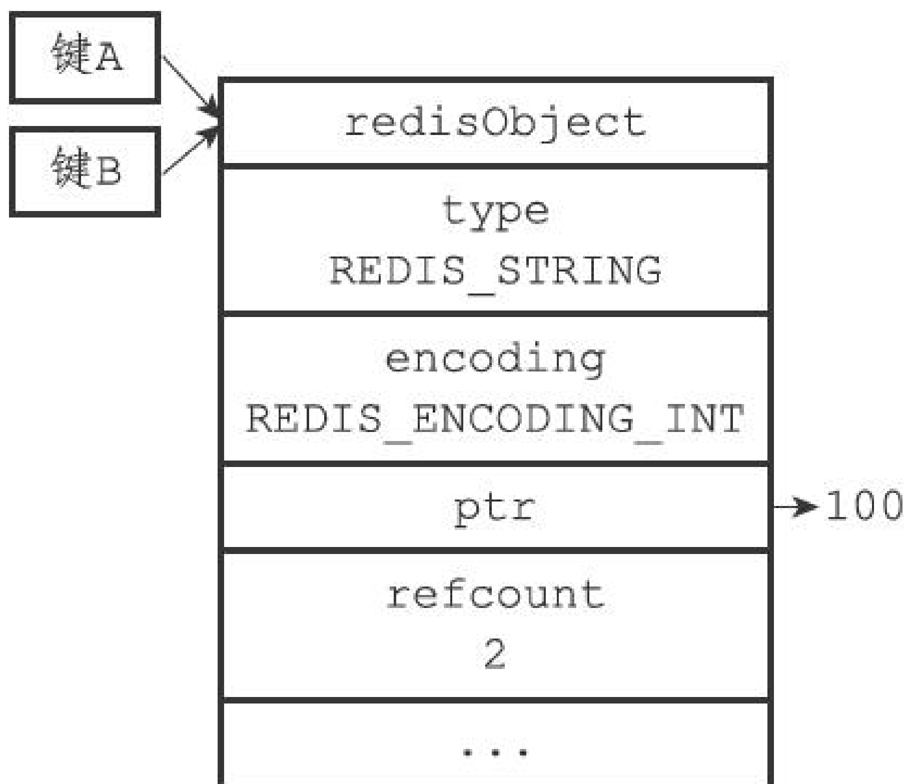


图8-21　被共享的字符串对象

例如，假设数据库中保存了整数值100的键不只有键A和键B两个，而是有一百个，那么服务器只需要用一个字符串对象的内存就可以保存原本需要使用一百个字符串对象的内存才能保存的数据。

目前来说，Redis会在初始化服务器时，创建一万个字符串对象，这些对象包含了从0到9999的所有整数值，当服务器需要用到值为0到9999的字符串对象时，服务器就会使用这些共享对象，而不是新创建对象。

注意

创建共享字符串对象的数量可以通过修改redis.h/REDIS_SHARED_INTEGERS常量来修改。

举个例子，如果我们创建一个值为100的键A，并使用OBJECT REFCOUNT命令查看键A的值对象的引用计数，我们会发现值对象的引用计数为2：

```
redis> SET A 100
OK
redis> OBJECT REFCOUNT A
(integer) 2
```

引用这个值对象的两个程序分别是持有这个值对象的服务器程序，以及共享这个值对象的键A，如图8-22所示。

如果这时我们再创建一个值为100的键B，那么键B也会指向包含整数值100的共享对象，使得共享对象的引用计数值变为3：

```
redis> SET B 100
OK
redis> OBJECT REFCOUNT A
(integer) 3
redis> OBJECT REFCOUNT B
(integer) 3
```

图8-23展示了共享值对象的三个程序。

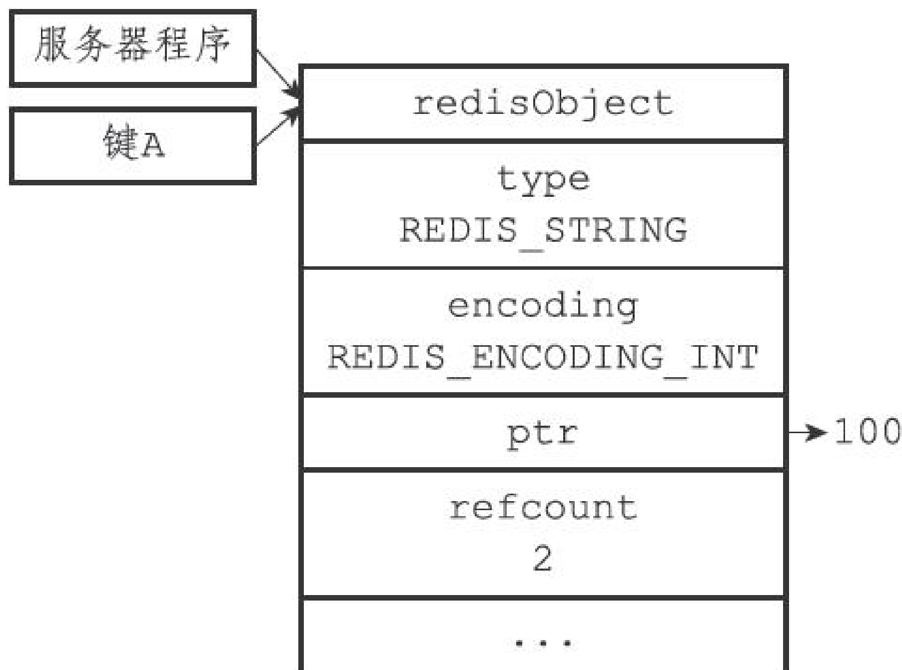


图8-22　引用数为2的共享对象

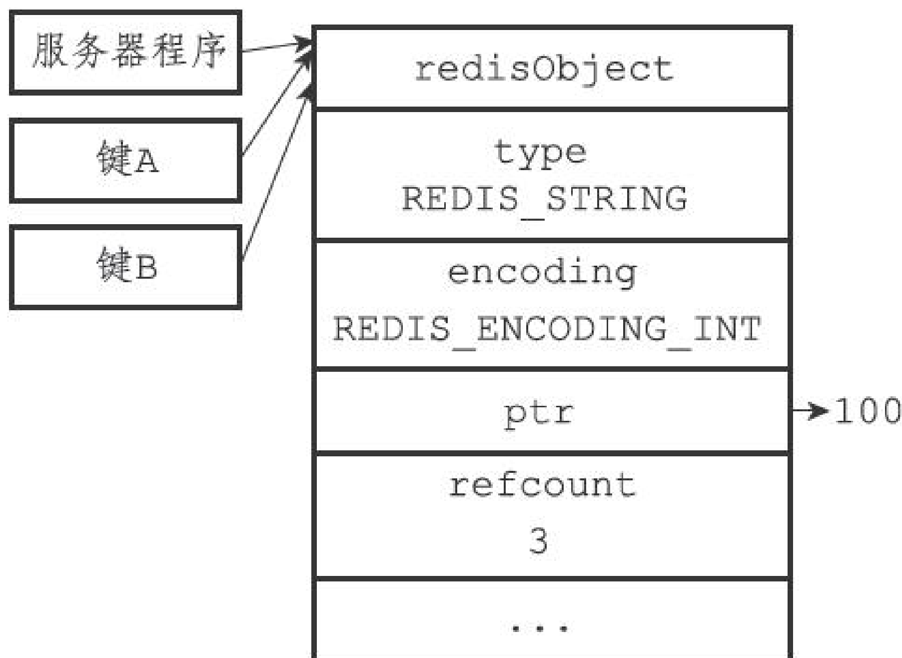


图8-23　引用数为3的共享对象

另外，这些共享对象不单单只有字符串键可以使用，那些在数据结构中嵌套了字符串对象的对象（linkedlist编码的列表对象、hashtable编码的哈希对象、hashtable编码的集合对象，以及zset编码的有序集合对

象）都可以使用这些共享对象。

为什么Redis不共享包含字符串的对象？

当服务器考虑将一个共享对象设置为键的值对象时，程序需要先检查给定的共享对象和键想创建的目标对象是否完全相同，只有在共享对象和目标对象完全相同的情况下，程序才会将共享对象用作键的值对象，而一个共享对象保存的值越复杂，验证共享对象和目标对象是否相同所需的复杂度就会越高，消耗的CPU时间也会越多：

·如果共享对象是保存整数值的字符串对象，那么验证操作的复杂度为O（1）；

·如果共享对象是保存字符串值的字符串对象，那么验证操作的复杂度为O（N）；

·如果共享对象是包含了多个值（或者对象的）对象，比如列表对象或者哈希对象，那么验证操作的复杂度将会是O（N_2）。

因此，尽管共享更复杂的对象可以节约更多的内存，但受到CPU时间的限制，Redis只对包含整数值的字符串对象进行共享。

8.10　对象的空转时长

除了前面介绍过的type、encoding、ptr和refcount四个属性之外，redisObject结构包含的最后一个属性为lru属性，该属性记录了对象最后一次被命令程序访问的时间：

```
typedef struct redisObject {
    // ...
    unsigned lru:22;
    // ...
} robj;
```

OBJECT IDLETIME命令可以打印出给定键的空转时长，这一空转时长就是通过将当前时间减去键的值对象的lru时间计算得出的：

```
redis> SET msg "hello world"
OK
#
等待一小段时间
redis> OBJECT IDLETIME msg
(integer) 20
#
等待一阵子
redis> OBJECT IDLETIME msg
(integer) 180
#
访问msg
键的值
redis> GET msg
"hello world"
#
键处于活跃状态，空转时长为0
redis> OBJECT IDLETIME msg
(integer) 0
```


注意

OBJECT IDLETIME命令的实现是特殊的，这个命令在访问键的值对象时，不会修改值对象的lru属性。

除了可以被OBJECT IDLETIME命令打印出来之外，键的空转时长还有另外一项作用：如果服务器打开了maxmemory选项，并且服务器用于回收内存的算法为volatile-lru或者allkeys-lru，那么当服务器占用的内存数超过了maxmemory选项所设置的上限值时，空转时长较高的那部分键会优先被服务器释放，从而回收内存。

配置文件的maxmemory选项和maxmemory-policy选项的说明介绍了关于这方面的更多信息。

8.11 重点回顾

·Redis数据库中的每个键值对的键和值都是一个对象。

·Redis共有字符串、列表、哈希、集合、有序集合五种类型的对象，每种类型的对象至少都有两种或以上的编码方式，不同的编码可以在不同的使用场景上优化对象的使用效率。

·服务器在执行某些命令之前，会先检查给定键的类型能否执行指定的命令，而检查一个键的类型就是检查键的值对象的类型。

·Redis的对象系统带有引用计数实现的内存回收机制，当一个对象不再被使用时，该对象所占用的内存就会被自动释放。

·Redis会共享值为0到9999的字符串对象。

·对象会记录自己的最后一次被访问的时间，这个时间可以用于计算对象的空转时间。

第二部分　单机数据库的实现

第9章　数据库

第10章　RDB持久化

第11章　AOF持久化

第12章　事件

第13章　客户端

第14章　服务器

第9章　数据库

本章将对Redis服务器的数据库实现进行详细介绍，说明服务器保存数据库的方法，客户端切换数据库的方法，数据库保存键值对的方法，以及针对数据库的添加、删除、查看、更新操作的实现方法等。除此之外，本章还会说明服务器保存键的过期时间的方法，以及服务器自动删除过期键的方法。最后，本章还会说明Redis 2.8新引入的数据库通知功能的实现方法。

9.1 服务器中的数据库

Redis服务器将所有数据库都保存在服务器状态redis.h/redisServer结构的db数组中，db数组的每个项都是一个redis.h/redisDb结构，每个redisDb结构代表一个数据库：

```
struct redisServer {
    // ...
    //
一个数组，保存着服务器中的所有数据库
    redisDb *db;
    // ...
};
```

在初始化服务器时，程序会根据服务器状态的dbnum属性来决定应该创建多少个数据库：

```
struct redisServer {
    // ...
    //
服务器的数据库数量
    int dbnum;
    // ...
};
```

dbnum属性的值由服务器配置的database选项决定，默认情况下，该选项的值为16，所以Redis服务器默认会创建16个数据库，如图9-1所示。

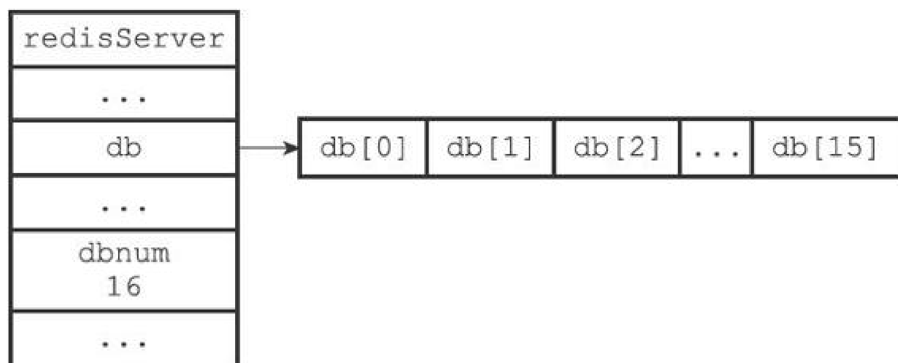


图9-1　服务器数据库示例

9.2 切换数据库

每个Redis客户端都有自己的目标数据库，每当客户端执行数据库写命令或者数据库读命令的时候，目标数据库就会成为这些命令的操作对象。

默认情况下，Redis客户端的目标数据库为0号数据库，但客户端可以通过执行SELECT命令来切换目标数据库。

以下代码示例演示了客户端在0号数据库设置并读取键msg，之后切换到2号数据库并执行类似操作的过程：

```
redis> SET msg "hello world"
OK
redis> GET msg
"hello world"
redis> SELECT 2
OK
redis[2]> GET msg
(nil)
redis[2]> SET msg"another world"
OK
redis[2]> GET msg
"another world"
```

在服务器内部，客户端状态redisClient结构的db属性记录了客户端当前的目标数据库，这个属性是一个指向redisDb结构的指针：

```
typedef struct redisClient {
// ...
//
记录客户端当前正在使用的数据库
redisDb *db;
// ...
} redisClient;
```

redisClient.db指针指向redisServer.db数组的其中一个元素，而被指向的元素就是客户端的目标数据库。

比如说，如果某个客户端的目标数据库为1号数据库，那么这个客户端所对应的客户端状态和服务器状态之间的关系如图9-2所示。

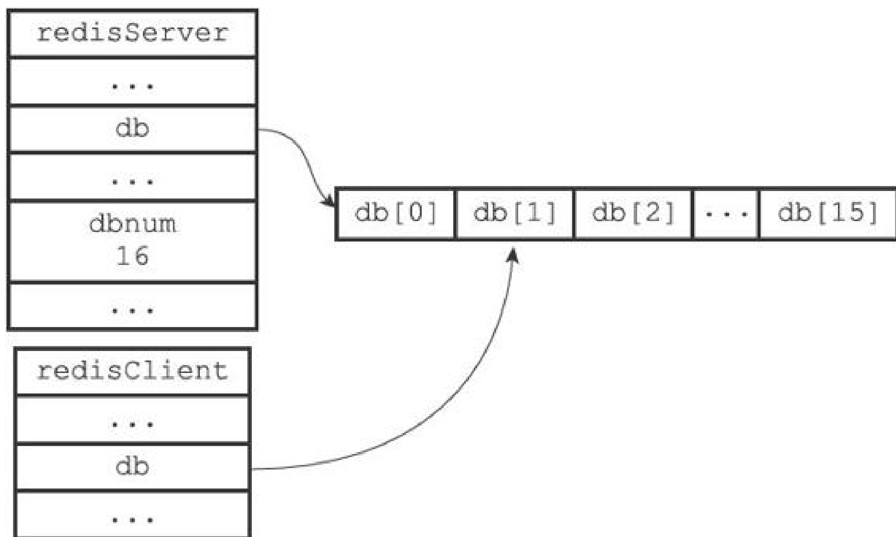


图9-2　客户端的目标数据库为1号数据库

如果这时客户端执行命令SELECT　　2，将目标数据库改为2号数据库，那么客户端状态和服务器状态之间的关系将更新成图9-3。

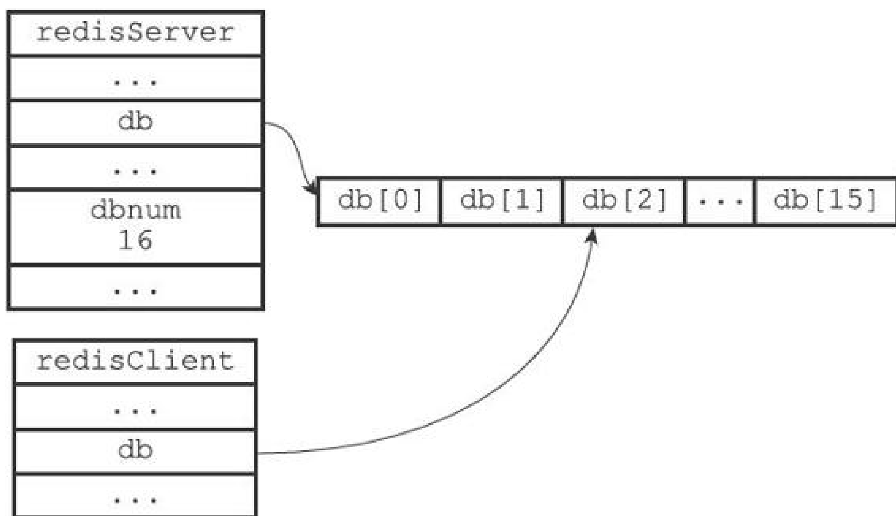


图9-3　客户端的目标数据库为2号数据库

通过修改redisClient.db指针，让它指向服务器中的不同数据库，从而实现切换目标数据库的功能——这就是SELECT命令的实现原理。

谨慎处理多数据库程序

到目前为止，Redis仍然没有可以返回客户端目标数据库的命

令。虽然redis-cli客户端会在输入符旁边提示当前所使用的目标数据库：

```
redis> SELECT 1
OK
redis[1]> SELECT 2
OK
redis[2]>
```

但如果你在其他语言的客户端中执行Redis命令，并且该客户端没有像redis-cli那样一直显示目标数据库的号码，那么在数次切换数据库之后，你很可能会忘记自己当前正在使用的是哪个数据库。当出现这种情况时，为了避免对数据库进行误操作，在执行Redis命令特别是像FLUSHDB这样的危险命令之前，最好先执行一个SELECT命令，显式地切换到指定的数据库，然后才执行别的命令。

9.3 数据库键空间

Redis是一个键值对（key-value pair）数据库服务器，服务器中的每个数据库都由一个redis.h/redisDb结构表示，其中，redisDb结构的dict字典保存了数据库中的所有键值对，我们将这个字典称为键空间（key space）：

```
typedef struct redisDb {
    // ...
    //
数据库键空间，保存着数据库中的所有键值对
    dict *dict;
    // ...
} redisDb;
```

键空间和用户所见的数据库是直接对应的：

·键空间的键也就是数据库的键，每个键都是一个字符串对象。

·键空间的值也就是数据库的值，每个值可以是字符串对象、列表对象、哈希表对象、集合对象和有序集合对象中的任意一种Redis对象。

举个例子，如果我们在空白的数据库中执行以下命令：

```
redis> SET message "hello world"
OK
redis> RPUSH alphabet "a" "b" "c"
(integer)3
redis> HSET book name "Redis in Action"
(integer) 1
redis> HSET book author "Josiah L. Carlson"
(integer) 1
redis> HSET book publisher "Manning"
(integer) 1
```

那么在这些命令执行之后，数据库的键空间将会是图9-4所展示的样子：

·alphabet是一个列表键，键的名字是一个包含字符串"alphabet"的字符串对象，键的值则是一个包含三个元素的列表对象。

·book是一个哈希表键，键的名字是一个包含字符串"book"的字符串对象，键的值则是一个包含三个键值对的哈希表对象。

·message是一个字符串键，键的名字是一个包含字符串"message"的字符串对象，键的值则是一个包含字符串"hello world"的字符串对象。

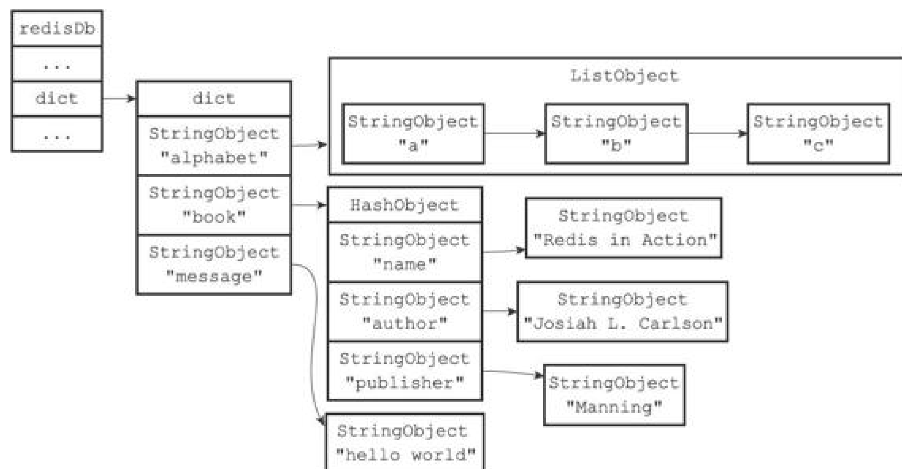


图9-4　数据库键空间例子

因为数据库的键空间是一个字典，所以所有针对数据库的操作，比如添加一个键值对到数据库，或者从数据库中删除一个键值对，又或者在数据库中获取某个键值对等，实际上都是通过对键空间字典进行操作来实现的，以下几个小节将分别介绍数据库的添加、删除、更新、取值等操作的实现原理。

9.3.1　添加新键

添加一个新键值对到数据库，实际上就是将一个新键值对添加到键空间字典里面，其中键为字符串对象，而值则为任意一种类型的Redis对象。

举个例子，如果键空间当前的状态如图9-4所示，那么在执行以下命令之后：

```
redis> SET date "2013.12.1"
OK
```

键空间将添加一个新的键值对，这个新键值对的键是一个包含字符串"date"的字符串对象，而键值对的值则是一个包含字符串"2013.12.1"的字符串对象，如图9-5所示。

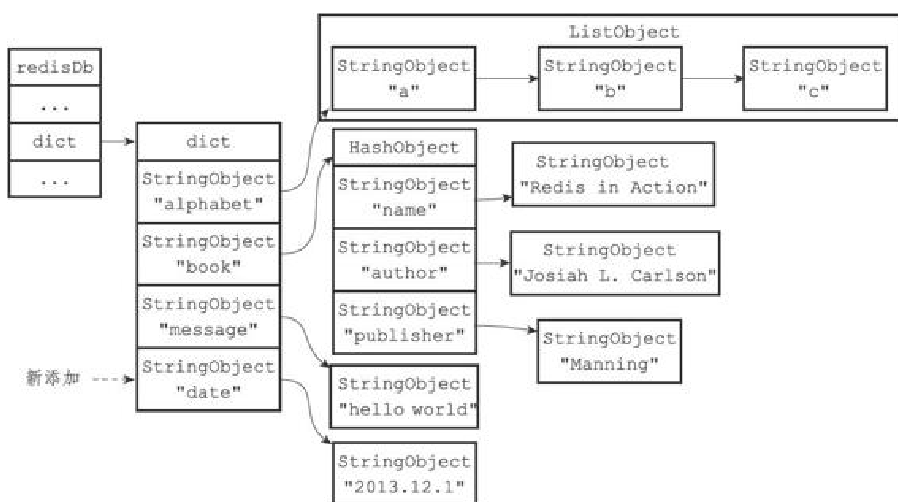


图9-5　添加date键之后的键空间

9.3.2　删除键

删除数据库中的一个键，实际上就是在键空间里面删除键所对应的键值对对象。

举个例子，如果键空间当前的状态如图9-4所示，那么在执行以下命令之后：

```
redis> DEL book
(integer) 1
```

键book以及它的值将从键空间中被删除，如图9-6所示。

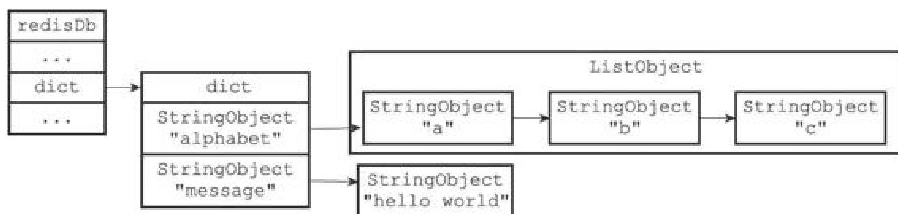


图9-6　删除book键之后的键空间

9.3.3　更新键

对一个数据库键进行更新，实际上就是对键空间里面键所对应的值对象进行更新，根据值对象的类型不同，更新的具体方法也会有所不同。

举个例子，如果键空间当前的状态如图9-4所示，那么在执行以下命令之后：

```
redis> SET message "blah blah"
OK
```

键message的值对象将从之前包含"hello world"字符串更新为包含"blah blah"字符串，如图9-7所示。

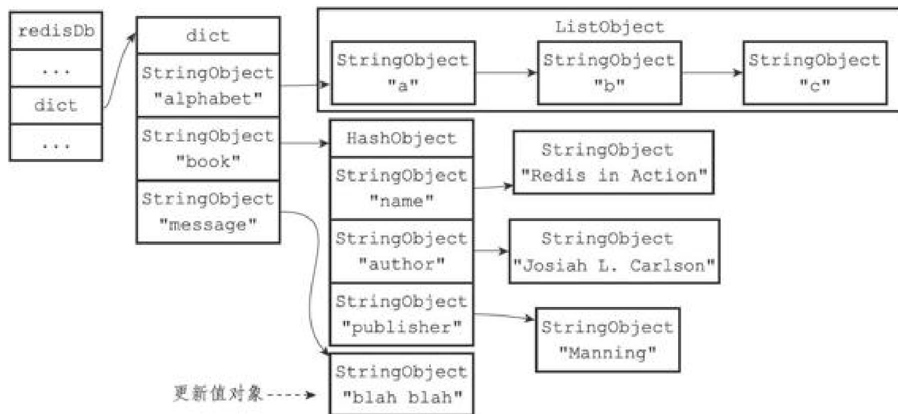


图9-7　使用SET命令更新message键

再举个例子，如果我们继续执行以下命令：

```
redis> HSET book page 320
(integer) 1
```

那么键空间中book键的值对象（一个哈希对象）将被更新，新的键值对page和320会被添加到值对象里面，如图9-8所示。

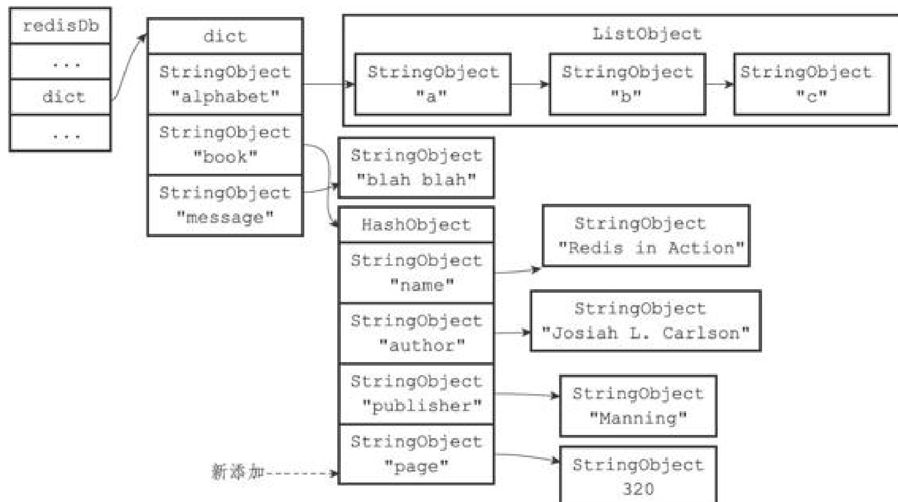


图9-8　使用HSET更新book键

9.3.4　对键取值

对一个数据库键进行取值，实际上就是在键空间中取出键所对应的值对象，根据值对象的类型不同，具体的取值方法也会有所不同。

举个例子，如果键空间当前的状态如图9-4所示，那么当执行以下命令时：

```
redis> GET message
"hello world"
```

GET命令将首先在键空间中查找键message，找到键之后接着取得该键所对应的字符串对象值，之后再返回值对象所包含的字符串"hello world"，取值过程如图9-9所示。

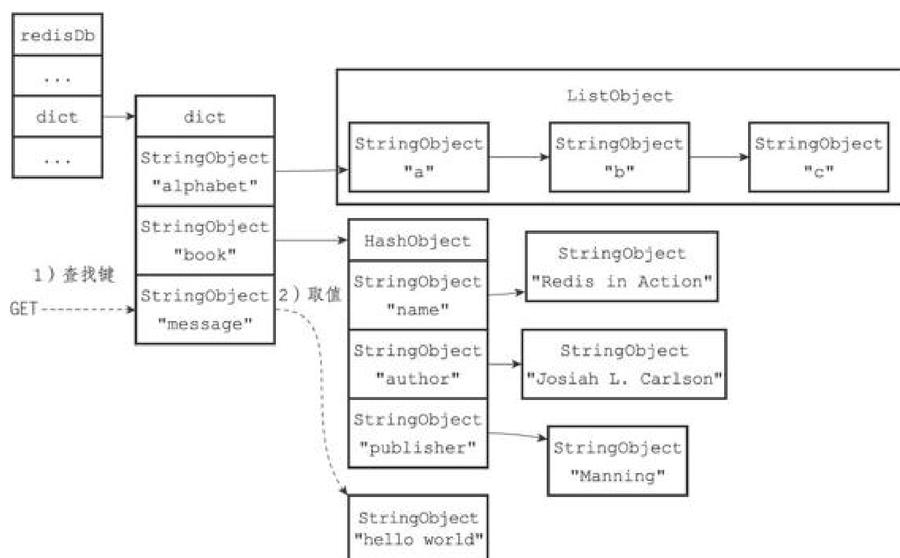


图9-9　使用GET命令取值的过程

再举一个例子，当执行以下命令时：

```
redis> LRANGE alphabet 0 -1
1) "a"
2) "b"
3) "c"
```

LRANGE命令将首先在键空间中查找键alphabet，找到键之后接着取得该键所对应的列表对象值，之后再返回列表对象中包含的三个字符串对象的值，取值过程如图9-10所示。

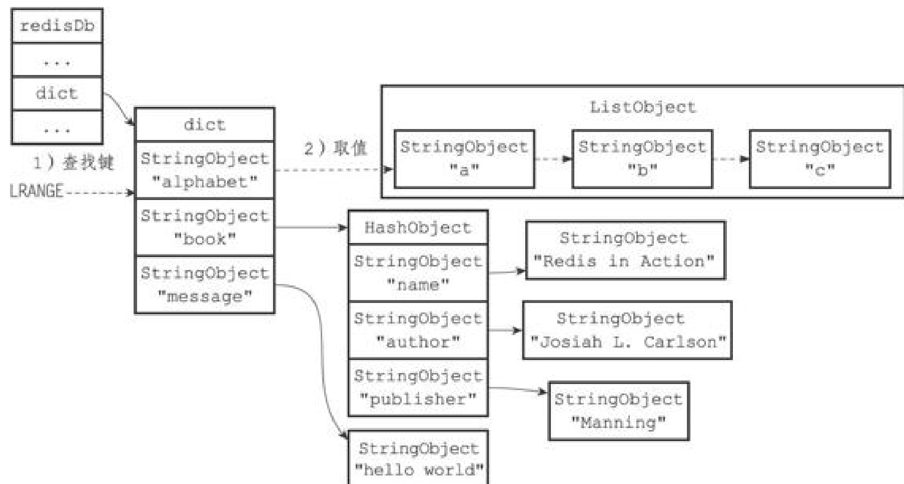


图9-10　使用LRANGE命令取值的过程

9.3.5　其他键空间操作

除了上面列出的添加、删除、更新、取值操作之外，还有很多针对数据库本身的Redis命令，也是通过对键空间进行处理来完成的。

比如说，用于清空整个数据库的FLUSHDB命令，就是通过删除键空间中的所有键值对来实现的。又比如说，用于随机返回数据库中某个键的RANDOMKEY命令，就是通过在键空间中随机返回一个键来实现的。

另外，用于返回数据库键数量的DBSIZE命令，就是通过返回键空间中包含的键值对的数量来实现的。类似的命令还有EXISTS、RENAME、KEYS等，这些命令都是通过对键空间进行操作来实现的。

9.3.6　读写键空间时的维护操作

当使用Redis命令对数据库进行读写时，服务器不仅会对键空间执行指定的读写操作，还会执行一些额外的维护操作，其中包括：

·在读取一个键之后（读操作和写操作都要对键进行读取），服务器会根据键是否存在来更新服务器的键空间命中（hit）次数或键空间不命中（miss）次数，这两个值可以在INFO　stats命令的keyspace_hits属性和keyspace_misses属性中查看。

·在读取一个键之后，服务器会更新键的LRU（最后一次使用）时间，这个值可以用于计算键的闲置时间，使用OBJECT　idletime命令可以查看键key的闲置时间。

·如果服务器在读取一个键时发现该键已经过期，那么服务器会先删除这个过期键，然后才执行余下的其他操作，本章稍后对过期键的讨论

会详细说明这一点。

·如果有客户端使用WATCH命令监视了某个键，那么服务器在对被监视的键进行修改之后，会将这个键标记为脏（dirty），从而让事务程序注意到这个键已经被修改过，第19章会详细说明这一点。

·服务器每次修改一个键之后，都会对脏（dirty）键计数器的值增1，这个计数器会触发服务器的持久化以及复制操作，第10章、第11章和第15章都会说到这一点。

·如果服务器开启了数据库通知功能，那么在对键进行修改之后，服务器将按配置发送相应的数据库通知，本章稍后讨论数据库通知功能的实现时会详细说明这一点。

9.4 设置键的生存时间或过期时间

通过EXPIRE命令或者PEXPIRE命令，客户端可以以秒或者毫秒精度为数据库中的某个键设置生存时间（Time To Live，TTL），在经过指定的秒数或者毫秒数之后，服务器就会自动删除生存时间为0的键：

```
redis> SET key value
OK
redis> EXPIRE key 5
(integer) 1
redis> GET key  // 5
秒之内
"value"
redis> GET key  // 5
秒之后
(nil)
```



注意

SETEX命令可以在设置一个字符串键的同时为键设置过期时间，因为这个命令是一个类型限定的命令（只能用于字符串键），所以本章不会对这个命令进行介绍，但SETEX命令设置过期时间的原理和本章介绍的EXPIRE命令设置过期时间的原理是完全一样的。

与EXPIRE命令和PEXPIRE命令类似，客户端可以通过EXPIREAT命令或PEXPIREAT命令，以秒或者毫秒精度给数据库中的某个键设置过期时间（expire time）。

过期时间是一个UNIX时间戳，当键的过期时间来临时，服务器就会自动从数据库中删除这个键：

```
redis> SET key value
OK
redis> EXPIREAT key 1377257300
(integer) 1
redis> TIME
1)"1377257296"
2)"296543"
redis> GET key    // 1377257300
之前
"value"
redis> TIME
1)"1377257303"
2)"230656"
redis> GET key    // 1377257300
之后
(nil)
```

TTL命令和PTTL命令接受一个带有生存时间或者过期时间的键，返

回这个键的剩余生存时间，也就是，返回距离这个键被服务器自动删除还有多长时间：

```
redis> SET key value
OK
redis> EXPIRE key 1000
(integer) 1
redis> TTL key
(integer) 997
redis> SET another_key another_value
OK
redis> TIME
1)"1377333070"
2)"761687"
redis> EXPIREAT another_key 1377333100
(integer) 1
redis> TTL another_key
(integer) 10
```

在上一节我们讨论了数据库的底层实现，以及各种数据库操作的实现原理，但是，关于数据库如何保存键的生存时间和过期时间，以及服务器如何自动删除那些带有生存时间和过期时间的键这两个问题，我们还没有讨论。

本节将对服务器保存键的生存时间和过期时间的方法进行介绍，并在下一节介绍服务器自动删除过期键的方法。

9.4.1 设置过期时间

Redis有四个不同的命令可以用于设置键的生存时间（键可以存在多久）或过期时间（键什么时候会被删除）：

·EXPIRE＜key＞＜ttl＞命令用于将键key的生存时间设置为ttl秒。

·PEXPIRE＜key＞＜ttl＞命令用于将键key的生存时间设置为ttl毫秒。

·EXPIREAT＜key＞＜timestamp＞命令用于将键key的过期时间设置为timestamp所指定的秒数时间戳。

·PEXPIREAT＜key＞＜timestamp＞命令用于将键key的过期时间设置为timestamp所指定的毫秒数时间戳。

虽然有多种不同单位和不同形式的设置命令，但实际上EXPIRE、PEXPIRE、EXPIREAT三个命令都是使用PEXPIREAT命令来实现的：无论客户端执行的是以上四个命令中的哪一个，经过转换之后，最终的执行效果都和执行PEXPIREAT命令一样。

首先，EXPIRE命令可以转换成PEXPIRE命令：

```
def EXPIRE(key,ttl_in_sec):
```

```
    #
将TTL
从秒转换成毫秒
    ttl_in_ms = sec_to_ms(ttl_in_sec)
    PEXPIRE(key, ttl_in_ms)
```

接着，PEXPIRE命令又可以转换成PEXPIREAT命令：

```
def PEXPIRE(key,ttl_in_ms):
    #
获取以毫秒计算的当前UNIX
时间戳
    now_ms = get_current_unix_timestamp_in_ms()
    #
当前时间加上TTL
，得出毫秒格式的键过期时间
    PEXPIREAT(key,now_ms+ttl_in_ms)
```

并且，EXPIREAT命令也可以转换成PEXPIREAT命令：

```
def EXPIREAT(key,expire_time_in_sec):
    #
将过期时间从秒转换为毫秒
    expire_time_in_ms = sec_to_ms(expire_time_in_sec)
    PEXPIREAT(key, expire_time_in_ms)
```

最终，EXPIRE、PEXPIRE和EXPIREAT三个命令都会转换成PEXPIREAT命令来执行，如图9-11所示。

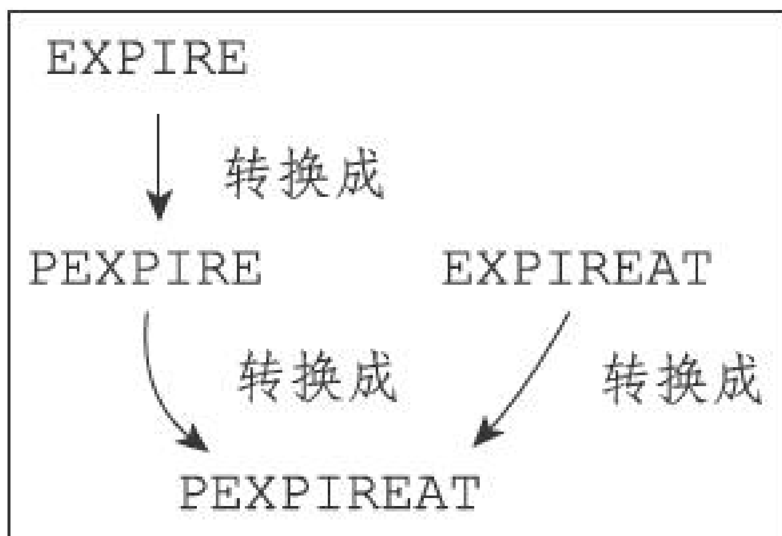


图9-11　设置生存时间和设置过期时间的命令之间的转换

9.4.2　保存过期时间

redisDb结构的expires字典保存了数据库中所有键的过期时间，我们称这个字典为过期字典：

·过期字典的键是一个指针，这个指针指向键空间中的某个键对象（也即是某个数据库键）。

·过期字典的值是一个long long类型的整数，这个整数保存了键所指向的数据库键的过期时间——一个毫秒精度的UNIX时间戳。

```
typedef struct redisDb {
    // ...
    //
过期字典，保存着键的过期时间
    dict *expires;
    // ...
} redisDb;
```

图9-12展示了一个带有过期字典的数据库例子，在这个例子中，键空间保存了数据库中的所有键值对，而过期字典则保存了数据库键的过期时间。



注意

为了展示方便，图9-12的键空间和过期字典中重复出现了两次alphabet键对象和book键对象。在实际中，键空间的键和过期字典的键都指向同一个键对象，所以不会出现任何重复对象，也不会浪费任何空间。

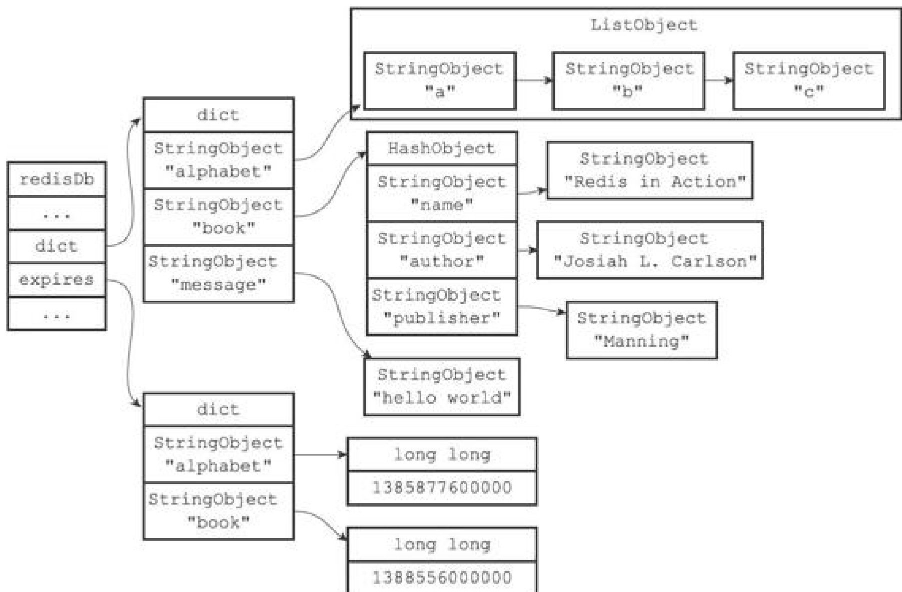


图9-12　带有过期字典的数据库例子

图9-12中的过期字典保存了两个键值对：

·第一个键值对的键为alphabet键对象，值为1385877600000，这表示数据库键alphabet的过期时间为1385877600000（2013年12月1日零时）。

·第二个键值对的键为book键对象，值为1388556000000，这表示数据库键book的过期时间为1388556000000（2014年1月1日零时）。

当客户端执行PEXPIREAT命令（或者其他三个会转换成PEXPIREAT命令的命令）为一个数据库键设置过期时间时，服务器会在数据库的过期字典中关联给定的数据库键和过期时间。

举个例子，如果数据库当前的状态如图9-12所示，那么在服务器执行以下命令之后：

```
redis> PEXPIREAT message 1391234400000
(integer) 1
```

过期字典将新增一个键值对，其中键为message键对象，而值则为1391234400000（2014年2月1日零时），如图9-13所示。

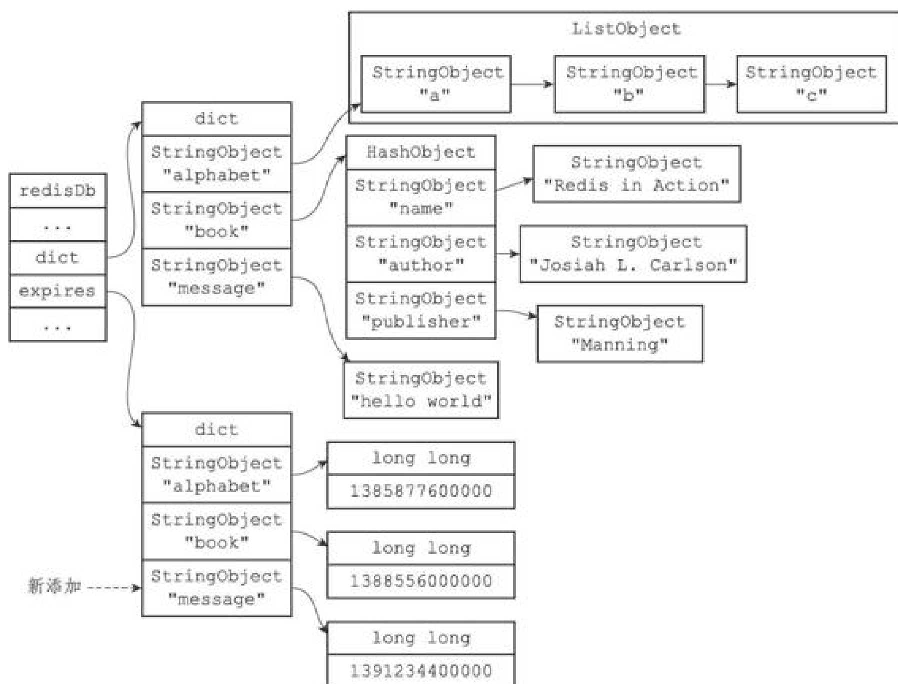


图9-13　执行PEXPIREAT命令之后的数据库

以下是PEXPIREAT命令的伪代码定义：

```
def PEXPIREAT(key, expire_time_in_ms):
    # 如果给定的键不存在于键空间，那么不能设置过期时间
    if key not in redisDb.dict:
        return0
    # 在过期字典中关联键和过期时间
    redisDb.expires[key] = expire_time_in_ms
    # 过期时间设置成功
    return 1
```

9.4.3 移除过期时间

PERSIST命令可以移除一个键的过期时间：

```
redis> PEXPIREAT message 1391234400000
(integer) 1
redis> TTL message
(integer) 13893281
redis> PERSIST message
(integer) 1
redis> TTL message
(integer) -1
```

PERSIST命令就是PEXPIREAT命令的反操作：PERSIST命令在过期字典中查找给定的键，并解除键和值（过期时间）在过期字典中的关联。

举个例子，如果数据库当前的状态如图9-12所示，那么当服务器执行以下命令之后：

```
redis> PERSIST book
(integer) 1
```

数据库将更新成图9-14所示的状态。

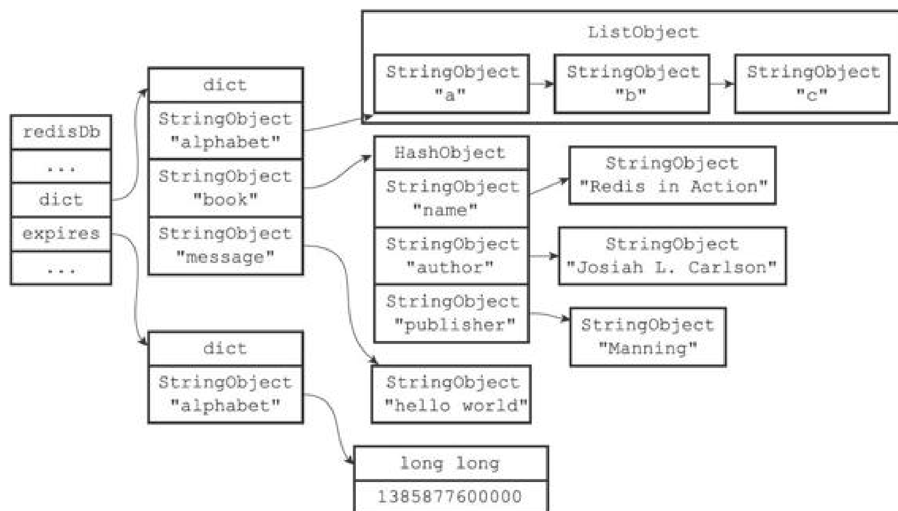


图9-14　执行PERSIST之后的数据库

可以看到，当PERSIST命令执行之后，过期字典中原来的book键值对消失了，这代表数据库键book的过期时间已经被移除。

以下是PERSIST命令的伪代码定义：

```
def PERSIST(key):
    #
如果键不存在，或者键没有设置过期时间，那么直接返回
    if key not in redisDb.expires:
        return0
    #
移除过期字典中给定键的键值对关联
    redisDb.expires.remove(key)
    #
键的过期时间移除成功
    return 1
```

9.4.4　计算并返回剩余生存时间

TTL命令以秒为单位返回键的剩余生存时间，而PTTL命令则以毫秒为单位返回键的剩余生存时间：

```
redis> PEXPIREAT alphabet 1385877600000
(integer) 1
redis> TTL alphabet
(integer) 8549007
redis> PTTL alphabet
(integer) 8549001011
```

TTL和PTTL两个命令都是通过计算键的过期时间和当前时间之间的差来实现的，以下是这两个命令的伪代码实现：

```
def PTTL(key):
    #
键不存在于数据库
    if key not in redisDb.dict:
        return-2
    #
尝试取得键的过期时间
    #
如果键没有设置过期时间，那么 expire_time_in_ms
将为 None
    expire_time_in_ms = redisDb.expires.get(key)
    #
键没有设置过期时间
    if expire_time_in_ms is None:
        return -1
    #
获得当前时间
    now_ms = get_current_unix_timestamp_in_ms()
    #
过期时间减去当前时间，得出的差就是键的剩余生存时间
    return(expire_time_in_ms - now_ms)
def TTL(key):
    #
获取以毫秒为单位的剩余生存时间
    ttl_in_ms = PTTL(key)
```

```
    if ttl_in_ms < 0:
        #
处理返回值为-2
和-1
的情况
        return ttl_in_ms
    else:
        #
将毫秒转换为秒
        return ms_to_sec(ttl_in_ms)
```

举个例子，对于一个过期时间为1385877600000（2013年12月1日零时）的键alphabet来说：

·如果当前时间为1383282000000（2013年11月1日零时），那么对键alphabet执行PTTL命令将返回2595600000，这个值是通过用alphabet键的过期时间减去当前时间计算得出的：1385877600000-1383282000000＝2595600000。

·另一方面，如果当前时间为1383282000000（2013年11月1日零时），那么对键alphabet执行TTL命令将返回2595600，这个值是通过计算alphabet键的过期时间减去当前时间的差，然后将差值从毫秒转换为秒之后得出的。

9.4.5　过期键的判定

通过过期字典，程序可以用以下步骤检查一个给定键是否过期：

1）检查给定键是否存在于过期字典：如果存在，那么取得键的过期时间。

2）检查当前UNIX时间戳是否大于键的过期时间：如果是的话，那么键已经过期；否则的话，键未过期。

可以用伪代码来描述这一过程：

```
def is_expired(key):
    #
取得键的过期时间
    expire_time_in_ms = redisDb.expires.get(key)
    #
键没有设置过期时间
    if expire_time_in_ms is None:
        return False
    #
取得当前时间的UNIX
时间戳
    now_ms = get_current_unix_timestamp_in_ms()
    #
检查当前时间是否大于键的过期时间
    if now_ms > expire_time_in_ms:
        #
是，键已经过期
        return True
    else:
        #
```

```
否，键未过期
        return False
```

举个例子，对于一个过期时间为1385877600000（2013年12月1日零时）的键alphabet来说：

·如果当前时间为1383282000000（2013年11月1日零时），那么调用is_expired（alphabet）将返回False，因为当前时间小于alphabet键的过期时间。

·另一方面，如果当前时间为1385964000000（2013年12月2日零时），那么调用is_expired（alphabet）将返回True，因为当前时间大于alphabet键的过期时间。

注意

实现过期键判定的另一种方法是使用TTL命令或者PTTL命令，比如说，如果对某个键执行TTL命令，并且命令返回的值大于等于0，那么说明该键未过期。在实际中，Redis检查键是否过期的方法和is_expired函数所描述的方法一致，因为直接访问字典比执行一个命令稍微快一些。

9.5 过期键删除策略

经过上一节的介绍，我们知道了数据库键的过期时间都保存在过期字典中，又知道了如何根据过期时间去判断一个键是否过期，现在剩下的问题是：如果一个键过期了，那么它什么时候会被删除呢？

这个问题有三种可能的答案，它们分别代表了三种不同的删除策略：

·定时删除：在设置键的过期时间的同时，创建一个定时器（timer），让定时器在键的过期时间来临时，立即执行对键的删除操作。

·惰性删除：放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键；如果没有过期，就返回该键。

·定期删除：每隔一段时间，程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

在这三种策略中，第一种和第三种为主动删除策略，而第二种则为被动删除策略。

9.5.1 定时删除

定时删除策略对内存是最友好的：通过使用定时器，定时删除策略可以保证过期键会尽可能快地被删除，并释放过期键所占用的内存。

另一方面，定时删除策略的缺点是，它对CPU时间是最不友好的：在过期键比较多的情况下，删除过期键这一行为可能会占用相当一部分CPU时间，在内存不紧张但是CPU时间非常紧张的情况下，将CPU时间用在删除和当前任务无关的过期键上，无疑会对服务器的响应时间和吞吐量造成影响。

例如，如果正有大量的命令请求在等待服务器处理，并且服务器当前不缺少内存，那么服务器应该优先将CPU时间用在处理客户端的命令请求上面，而不是用在删除过期键上面。

除此之外，创建一个定时器需要用到Redis服务器中的时间事件，而当前时间事件的实现方式——无序链表，查找一个事件的时间复杂度为O（N）——并不能高效地处理大量时间事件。

因此，要让服务器创建大量的定时器，从而实现定时删除策略，在现阶段来说并不现实。

9.5.2 惰性删除

惰性删除策略对CPU时间来说是最友好的：程序只会在取出键时才对键进行过期检查，这可以保证删除过期键的操作只会在非做不可的情况下进行，并且删除的目标仅限于当前处理的键，这个策略不会在删除其他无关的过期键上花费任何CPU时间。

惰性删除策略的缺点是，它对内存是最不友好的：如果一个键已经过期，而这个键又仍然保留在数据库中，那么只要这个过期键不被删除，它所占用的内存就不会释放。

在使用惰性删除策略时，如果数据库中有非常多的过期键，而这些过期键又恰好没有被访问到的话，那么它们也许永远也不会被删除（除非用户手动执行FLUSHDB），我们甚至可以将这种情况看作是一种内存泄漏——无用的垃圾数据占用了大量的内存，而服务器却不会自己去释放它们，这对于运行状态非常依赖于内存的Redis服务器来说，肯定不是一个好消息。

举个例子，对于一些和时间有关的数据，比如日志（log），在某个时间点之后，对它们的访问就会大大减少，甚至不再访问，如果这类过期数据大量地积压在数据库中，用户以为服务器已经自动将它们删除了，但实际上这些键仍然存在，而且键所占用的内存也没有释放，那么造成的后果肯定是非常严重的。

9.5.3 定期删除

从上面对定时删除和惰性删除的讨论来看，这两种删除方式在单一使用时都有明显的缺陷：

·定时删除占用太多CPU时间，影响服务器的响应时间和吞吐量。

·惰性删除浪费太多内存，有内存泄漏的危险。

定期删除策略是前两种策略的一种整合和折中：

·定期删除策略每隔一段时间执行一次删除过期键操作，并通过限制删除操作执行的时长和频率来减少删除操作对CPU时间的影响。

·除此之外，通过定期删除过期键，定期删除策略有效地减少了因为过期键而带来的内存浪费。

定期删除策略的难点是确定删除操作执行的时长和频率：

·如果删除操作执行得太频繁，或者执行的时间太长，定期删除策略就会退化成定时删除策略，以至于将CPU时间过多地消耗在删除过期键上面。

·如果删除操作执行得太少，或者执行的时间太短，定期删除策略又会和惰性删除策略一样，出现浪费内存的情况。

因此，如果采用定期删除策略的话，服务器必须根据情况，合理地设置删除操作的执行时长和执行频率。

9.6 Redis的过期键删除策略

在前一节，我们讨论了定时删除、惰性删除和定期删除三种过期键删除策略，Redis服务器实际使用的是惰性删除和定期删除两种策略：通过配合使用这两种删除策略，服务器可以很好地在合理使用CPU时间和避免浪费内存空间之间取得平衡。

因为前一节已经介绍过惰性删除和定期删除两种策略的概念了，在接下来的两个小节中，我们将对Redis服务器中惰性删除和定期删除的具体实现进行说明。

9.6.1 惰性删除策略的实现

过期键的惰性删除策略由db.c/expireIfNeeded函数实现，所有读写数据库的Redis命令在执行之前都会调用expireIfNeeded函数对输入键进行检查：

·如果输入键已经过期，那么expireIfNeeded函数将输入键从数据库中删除。

·如果输入键未过期，那么expireIfNeeded函数不做动作。

命令调用expireIfNeeded函数的过程如图9-15所示。

expireIfNeeded函数就像一个过滤器，它可以在命令真正执行之前，过滤掉过期的输入键，从而避免命令接触到过期键。

另外，因为每个被访问的键都可能因为过期而被expireIfNeeded函数删除，所以每个命令的实现函数都必须能同时处理键存在以及键不存在这两种情况：

·当键存在时，命令按照键存在的情况执行。

·当键不存在或者键因为过期而被expireIfNeeded函数删除时，命令按照键不存在的情况执行。

举个例子，图9-16展示了GET命令的执行过程，在这个执行过程中，命令需要判断键是否存在以及键是否过期，然后根据判断来执行合适的动作。

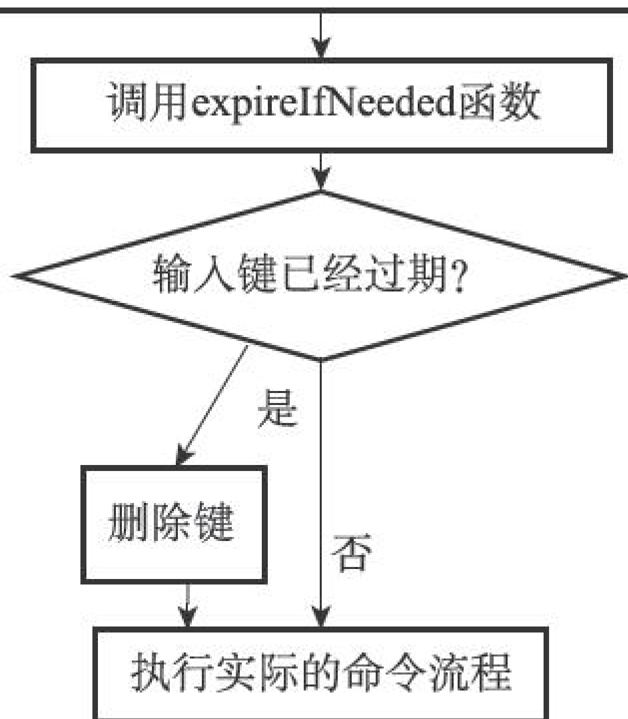


图9-15　命令调用expireIfNeeded来删除过期键

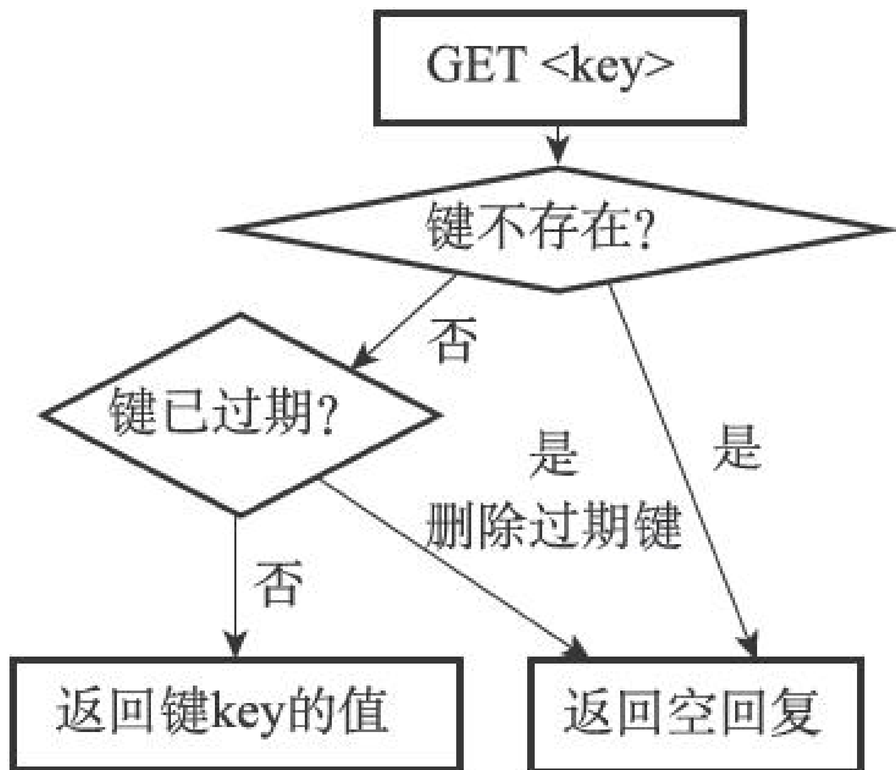


图9-16　GET命令的执行过程

9.6.2　定期删除策略的实现

过期键的定期删除策略由redis.c/activeExpireCycle函数实现，每当Redis的服务器周期性操作redis.c/serverCron函数执行时，activeExpireCycle函数就会被调用，它在规定的时间内，分多次遍历服务器中的各个数据库，从数据库的expires字典中随机检查一部分键的过期时间，并删除其中的过期键。

整个过程可以用伪代码描述如下：

```
#
默认每次检查的数据库数量
DEFAULT_DB_NUMBERS = 16
#
默认每个数据库检查的键数量
DEFAULT_KEY_NUMBERS = 20
#
全局变量，记录检查进度
current_db = 0
def activeExpireCycle():
    #
初始化要检查的数据库数量
    #
如果服务器的数据库数量比 DEFAULT_DB_NUMBERS
要小
    #
```

```
那么以服务器的数据库数量为准
    if server.dbnum < DEFAULT_DB_NUMBERS:
        db_numbers = server.dbnum
    else:
        db_numbers = DEFAULT_DB_NUMBERS
    #
遍历各个数据库
    for i in range(db_numbers):
        #
如果current_db
的值等于服务器的数据库数量
        #
这表示检查程序已经遍历了服务器的所有数据库一次
        #
将current_db
重置为0
，开始新的一轮遍历
        if current_db == server.dbnum:
            current_db = 0
        #
获取当前要处理的数据库
        redisDb = server.db[current_db]
        #
将数据库索引增1
，指向下一个要处理的数据库
        current_db += 1
        #
检查数据库键
        for j in range(DEFAULT_KEY_NUMBERS):
            #
如果数据库中没有一个键带有过期时间，那么跳过这个数据库
            if redisDb.expires.size() == 0: break
            #
随机获取一个带有过期时间的键
            key_with_ttl = redisDb.expires.get_random_key()
            #
检查键是否过期，如果过期就删除它
            if is_expired(key_with_ttl):
                delete_key(key_with_ttl)
            #
已达到时间上限，停止处理
            if reach_time_limit(): return
```

activeExpireCycle函数的工作模式可以总结如下：

·函数每次运行时，都从一定数量的数据库中取出一定数量的随机键进行检查，并删除其中的过期键。

·全局变量current_db会记录当前activeExpireCycle函数检查的进度，并在下一次activeExpireCycle函数调用时，接着上一次的进度进行处理。比如说，如果当前activeExpireCycle函数在遍历10号数据库时返回了，那么下次activeExpireCycle函数执行时，将从11号数据库开始查找并删除过期键。

·随着activeExpireCycle函数的不断执行，服务器中的所有数据库都会被检查一遍，这时函数将current_db变量重置为0，然后再次开始新一轮的检查工作。

9.7 AOF、RDB和复制功能对过期键的处理

在这一节，我们将探讨过期键对Redis服务器中其他模块的影响，看看RDB持久化功能、AOF持久化功能以及复制功能是如何处理数据库中的过期键的。

9.7.1 生成RDB文件

在执行SAVE命令或者BGSAVE命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存到新创建的RDB文件中。

举个例子，如果数据库中包含三个键k1、k2、k3，并且k2已经过期，那么当执行SAVE命令或者BGSAVE命令时，程序只会将k1和k3的数据保存到RDB文件中，而k2则会被忽略。

因此，数据库中包含过期键不会对生成新的RDB文件造成影响。

9.7.2 载入RDB文件

在启动Redis服务器时，如果服务器开启了RDB功能，那么服务器将对RDB文件进行载入：

·如果服务器以主服务器模式运行，那么在载入RDB文件时，程序会对文件中保存的键进行检查，未过期的键会被载入到数据库中，而过期键则会被忽略，所以过期键对载入RDB文件的主服务器不会造成影响。

·如果服务器以从服务器模式运行，那么在载入RDB文件时，文件中保存的所有键，不论是否过期，都会被载入到数据库中。不过，因为主从服务器在进行数据同步的时候，从服务器的数据库就会被清空，所以一般来讲，过期键对载入RDB文件的从服务器也不会造成影响。

举个例子，如果数据库中包含三个键k1、k2、k3，并且k2已经过期，那么当服务器启动时：

·如果服务器以主服务器模式运行，那么程序只会将k1和k3载入到数据库，k2会被忽略。

·如果服务器以从服务器模式运行，那么k1、k2和k3都会被载入到数据库。

9.7.3 AOF文件写入

当服务器以AOF持久化模式运行时，如果数据库中的某个键已经过期，但它还没有被惰性删除或者定期删除，那么AOF文件不会因为这个

过期键而产生任何影响。

当过期键被惰性删除或者定期删除之后，程序会向AOF文件追加（append）一条DEL命令，来显式地记录该键已被删除。

举个例子，如果客户端使用GET message命令，试图访问过期的message键，那么服务器将执行以下三个动作：

1）从数据库中删除message键。

2）追加一条DEL message命令到AOF文件。

3）向执行GET命令的客户端返回空回复。

9.7.4 AOF重写

和生成RDB文件时类似，在执行AOF重写的过程中，程序会对数据库中的键进行检查，已过期的键不会被保存到重写后的AOF文件中。

举个例子，如果数据库中包含三个键k1、k2、k3，并且k2已经过期，那么在进行重写工作时，程序只会对k1和k3进行重写，而k2则会被忽略。

因此，数据库中包含过期键不会对AOF重写造成影响。

9.7.5 复制

当服务器运行在复制模式下时，从服务器的过期键删除动作由主服务器控制：

·主服务器在删除一个过期键之后，会显式地向所有从服务器发送一个DEL命令，告知从服务器删除这个过期键。

·从服务器在执行客户端发送的读命令时，即使碰到过期键也不会将过期键删除，而是继续像处理未过期的键一样来处理过期键。

·从服务器只有在接到主服务器发来的DEL命令之后，才会删除过期键。

通过由主服务器来控制从服务器统一地删除过期键，可以保证主从服务器数据的一致性，也正是因为这个原因，当一个过期键仍然存在于主服务器的数据库时，这个过期键在从服务器里的复制品也会继续存在。

举个例子，有一对主从服务器，它们的数据库中都保存着同样的三个键message、xxx和yyy，其中message为过期键，如图9-17所示。

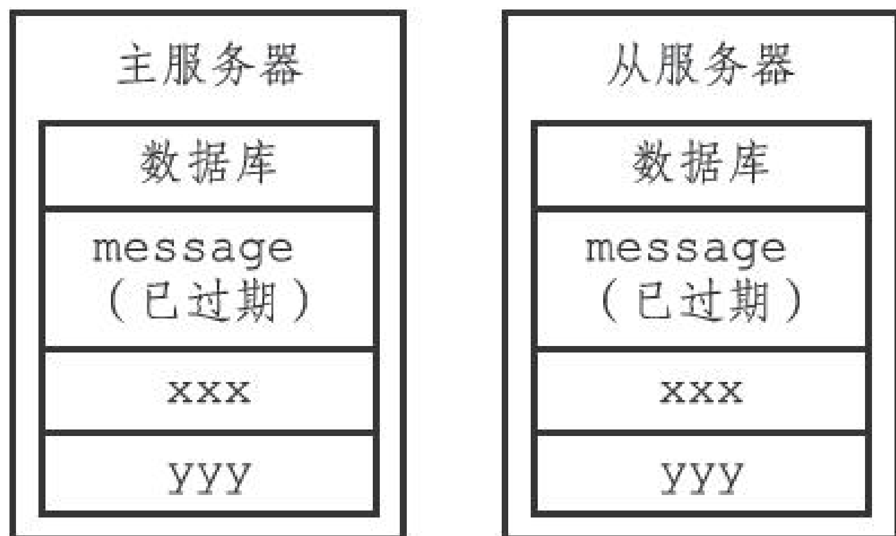


图9-17　主从服务器删除过期键（1）

如果这时有客户端向从服务器发送命令GET　message，那么从服务器将发现message键已经过期，但从服务器并不会删除message键，而是继续将message键的值返回给客户端，就好像message键并没有过期一样，如图9-18所示。

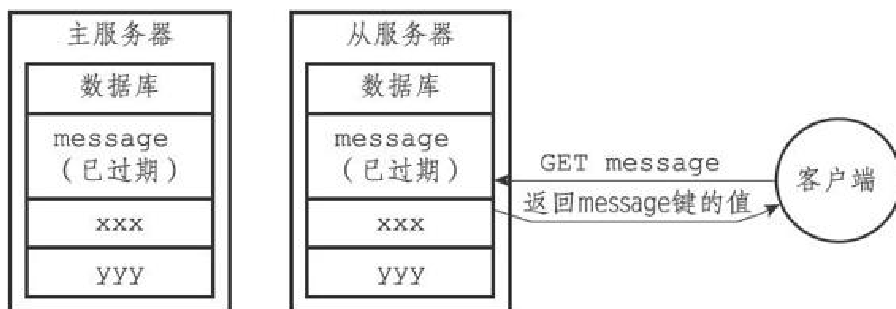


图9-18　主从服务器删除过期键（2）

假设在此之后，有客户端向主服务器发送命令GET　message，那么主服务器将发现键message已经过期：主服务器会删除message键，向客户端返回空回复，并向从服务器发送DEL　message命令，如图9-19所示。

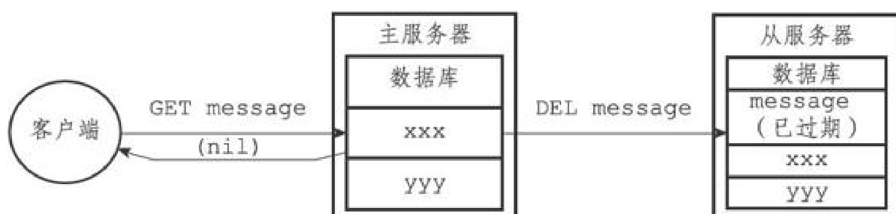


图9-19　主从服务器删除过期键（3）

从服务器在接收到主服务器发来的DEL　message命令之后，也会从数据库中删除message键，在这之后，主从服务器都不再保存过期键message了，如图9-20所示。

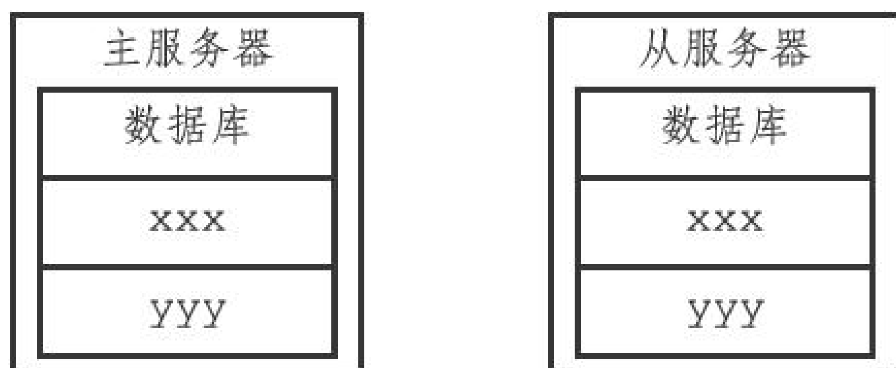


图9-20　主从服务器删除过期键（4）

9.8 数据库通知

数据库通知是Redis 2.8版本新增加的功能，这个功能可以让客户端通过订阅给定的频道或者模式，来获知数据库中键的变化，以及数据库中命令的执行情况。

举个例子，以下代码展示了客户端如何获取0号数据库中针对message键执行的所有命令：

```
127.0.0.1:6379> SUBSCRIBE _ _keyspace@0_ _:message
Reading messages... (press Ctrl-C to quit)
1) "subscribe"  //
订阅信息
2) "__keyspace@0__:message"
3) (integer) 1
1) "message"    //
执行SET
命令
2) "_ _keyspace@0_ _:message"
3) "set"
1) "message"    //
执行EXPIRE
命令
2) "_ _keyspace@0_ _:message"
3) "expire"
1) "message"    //
执行DEL
命令
2) "_ _keyspace@0_ _:message"
3) "del"
```

根据发回的通知显示，先后共有SET、EXPIRE、DEL三个命令对键message进行了操作。

这一类关注“某个键执行了什么命令”的通知称为键空间通知（key-space notification），除此之外，还有另一类称为键事件通知（key-event notification）的通知，它们关注的是“某个命令被什么键执行了”。

以下是一个键事件通知的例子，代码展示了客户端如何获取0号数据库中所有执行DEL命令的键：

```
127.0.0.1:6379> SUBSCRIBE _ _keyevent@0_ _:del
Reading messages... (press Ctrl-C to quit)
1) "subscribe"  //
订阅信息
2) "_ _keyevent@0_ _:del"
3) (integer) 1
1) "message"    //
键key
执行了DEL
命令
2) "_ _keyevent@0_ _:del"
3) "key"
1) "message"    //
键number
```

```
执行了DEL
命令
2) "_ _keyevent@0_ _:del"
3) "number"
1) "message"    //
键message
执行了DEL
命令
2) "_ _keyevent@0_ _:del"
3) "message"
```

根据发回的通知显示，key、number、message三个键先后执行了DEL命令。

服务器配置的notify-keyspace-events选项决定了服务器所发送通知的类型：

·想让服务器发送所有类型的键空间通知和键事件通知，可以将选项的值设置为AKE。

·想让服务器发送所有类型的键空间通知，可以将选项的值设置为AK。

·想让服务器发送所有类型的键事件通知，可以将选项的值设置为AE。

·想让服务器只发送和字符串键有关的键空间通知，可以将选项的值设置为K$。

·想让服务器只发送和列表键有关的键事件通知，可以将选项的值设置为El。

关于数据库通知功能的详细用法，以及notify-keyspace-events选项的更多设置，Redis的官方文档已经做了很详细的介绍，这里不再赘述。

在接下来的内容中，我们来看看数据库通知功能的实现原理。

9.8.1　发送通知

发送数据库通知的功能是由notify.c/notifyKeyspaceEvent函数实现的：

```
void notifyKeyspaceEvent(int type,char *event,robj *key,int dbid);
```

函数的type参数是当前想要发送的通知的类型，程序会根据这个值来判断通知是否就是服务器配置notify-keyspace-events选项所选定的通知类型，从而决定是否发送通知。

event、keys和dbid分别是事件的名称、产生事件的键，以及产生事件的数据库号码，函数会根据type参数以及这三个参数来构建事件通知

的内容，以及接收通知的频道名。

每当一个Redis命令需要发送数据库通知的时候，该命令的实现函数就会调用notify-KeyspaceEvent函数，并向函数传递传递该命令所引发的事件的相关信息。

例如，以下是SADD命令的实现函数saddCommand的其中一部分代码：

```
void saddCommand(redisClient*c){
    // ...
    //
如果至少有一个元素被成功添加，那么执行以下程序
    if (added) {
        // ...
        //
发送事件通知
        notifyKeyspaceEvent(REDIS_NOTIFY_SET,"sadd",c->argv[1],c->db->id);
    }
    // ...
}
```

当SADD命令至少成功地向集合添加了一个集合元素之后，命令就会发送通知，该通知的类型为REDIS_NOTIFY_SET（表示这是一个集合键通知），名称为sadd（表示这是执行SADD命令所产生的通知）。

以下是另一个例子，展示了DEL命令的实现函数delCommand的其中一部分代码：

```
voi delCommand(redisClient *c){
    int deleted=0,j;
    //
遍历所有输入键
    for (j=1; j<c->argc; j++){
        //
尝试删除键
        if (dbDelete(c->db,c->argv[j])){
            // ...
            //
删除键成功，发送通知
            notifyKeyspaceEvent(REDIS_NOTIFY_GENERIC,
                "del",c->argv[j],c->db->id);
            // ...
        }
    }
    // ...
}
```

在delCommand函数中，函数遍历所有输入键，并在删除键成功时，发送通知，通知的类型为REDIS_NOTIFY_GENERIC（表示这是一个通用类型的通知），名称为del（表示这是执行DEL命令所产生的通知）。

其他发送通知的函数调用notifyKeyspaceEvent函数的方式也和saddCommand、delCommand类似，只是给定的参数不同，接下来我们来看看notifyKeyspaceEvent函数的实现。

9.8.2　发送通知的实现

以下是notifyKeyspaceEvent函数的伪代码实现：

```
def notifyKeyspaceEvent(type, event, key, dbid):
    #
如果给定的通知不是服务器允许发送的通知，那么直接返回
    if not(server.notify_keyspace_events & type):
        return
    #
发送键空间通知
    if server.notify_keyspace_events & REDIS_NOTIFY_KEYSPACE:
        #
将通知发送给频道__keyspace@<dbid>__:<key>
        #
内容为键所发生的事件 <event>
        #
构建频道名字
        chan = "__keyspace@{dbid}__:{key}".format(dbid=dbid, key=key)
        #
发送通知
        pubsubPublishMessage(chan, event)
    #
发送键事件通知
    if server.notify_keyspace_events & REDIS_NOTIFY_KEYEVENT:
        #
将通知发送给频道__keyevent@<dbid>__:<event>
        #
内容为发生事件的键 <key>
        #
构建频道名字
        chan = "__keyevent@{dbid}__:{event}".format(dbid=dbid,event=event)
        #
发送通知
        pubsubPublishMessage(chan, key)
```

notifyKeyspaceEvent函数执行以下操作：

1）server.notify_keyspace_events属性就是服务器配置notify-keyspace-events选项所设置的值，如果给定的通知类型type不是服务器允许发送的通知类型，那么函数会直接返回，不做任何动作。

2）如果给定的通知是服务器允许发送的通知，那么下一步函数会检测服务器是否允许发送键空间通知，如果允许的话，程序就会构建并发送事件通知。

3）最后，函数检测服务器是否允许发送键事件通知，如果允许的话，程序就会构建并发送事件通知。

另外，pubsubPublishMessage函数是PUBLISH命令的实现函数，执行这个函数等同于执行PUBLISH命令，订阅数据库通知的客户端收到的信息就是由这个函数发出的，pubsubPublishMessage函数具体的实现细节可以参考第18章。

9.9 重点回顾

·Redis服务器的所有数据库都保存在redisServer.db数组中，而数据库的数量则由redisServer.dbnum属性保存。

·客户端通过修改目标数据库指针，让它指向redisServer.db数组中的不同元素来切换不同的数据库。

·数据库主要由dict和expires两个字典构成，其中dict字典负责保存键值对，而expires字典则负责保存键的过期时间。

·因为数据库由字典构成，所以对数据库的操作都是建立在字典操作之上的。

·数据库的键总是一个字符串对象，而值则可以是任意一种Redis对象类型，包括字符串对象、哈希表对象、集合对象、列表对象和有序集合对象，分别对应字符串键、哈希表键、集合键、列表键和有序集合键。

·expires字典的键指向数据库中的某个键，而值则记录了数据库键的过期时间，过期时间是一个以毫秒为单位的UNIX时间戳。

·Redis使用惰性删除和定期删除两种策略来删除过期的键：惰性删除策略只在碰到过期键时才进行删除操作，定期删除策略则每隔一段时间主动查找并删除过期键。

·执行SAVE命令或者BGSAVE命令所产生的新RDB文件不会包含已经过期的键。

·执行BGREWRITEAOF命令所产生的重写AOF文件不会包含已经过期的键。

·当一个过期键被删除之后，服务器会追加一条DEL命令到现有AOF文件的末尾，显式地删除过期键。

·当主服务器删除一个过期键之后，它会向所有从服务器发送一条DEL命令，显式地删除过期键。

·从服务器即使发现过期键也不会自作主张地删除它，而是等待主节点发来DEL命令，这种统一、中心化的过期键删除策略可以保证主从服务器数据的一致性。

·当Redis命令对数据库进行修改之后，服务器会根据配置向客户端发送数据库通知。

第10章　RDB持久化

Redis是一个键值对数据库服务器，服务器中通常包含着任意个非空数据库，而每个非空数据库中又可以包含任意个键值对，为了方便起见，我们将服务器中的非空数据库以及它们的键值对统称为数据库状态。

举个例子，图10-1展示了一个包含三个非空数据库的Redis服务器，这三个数据库以及数据库中的键值对就是该服务器的数据库状态。

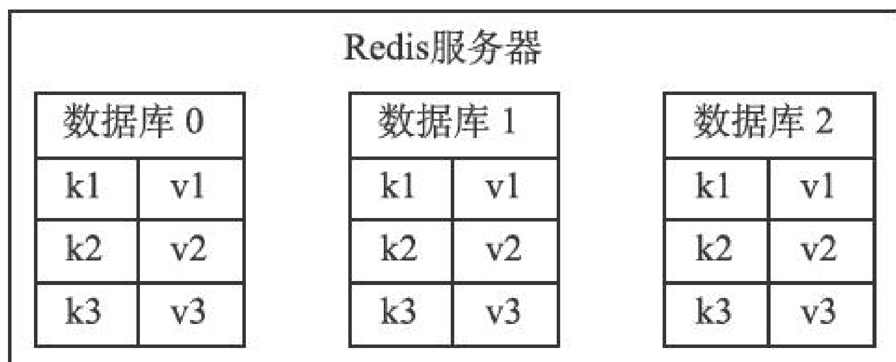


图10-1　数据库状态示例

因为Redis是内存数据库，它将自己的数据库状态储存在内存里面，所以如果不想办法将储存在内存中的数据库状态保存到磁盘里面，那么一旦服务器进程退出，服务器中的数据库状态也会消失不见。

为了解决这个问题，Redis提供了RDB持久化功能，这个功能可以将Redis在内存中的数据库状态保存到磁盘里面，避免数据意外丢失。

RDB持久化既可以手动执行，也可以根据服务器配置选项定期执行，该功能可以将某个时间点上的数据库状态保存到一个RDB文件中，如图10-2所示。

RDB持久化功能所生成的RDB文件是一个经过压缩的二进制文件，通过该文件可以还原生成RDB文件时的数据库状态，如图10-3所示。

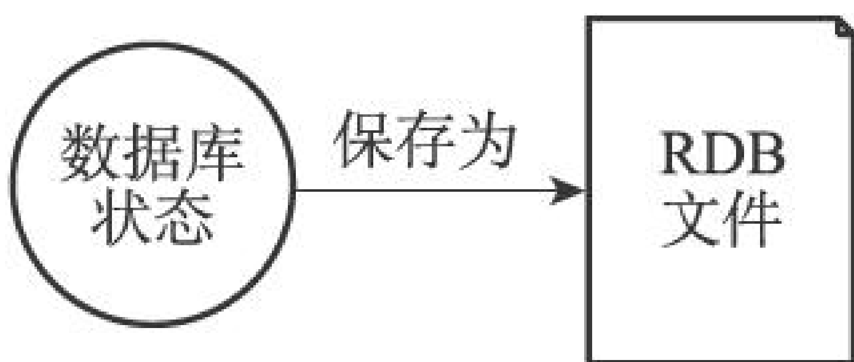


图10-2　将数据库状态保存为RDB文件

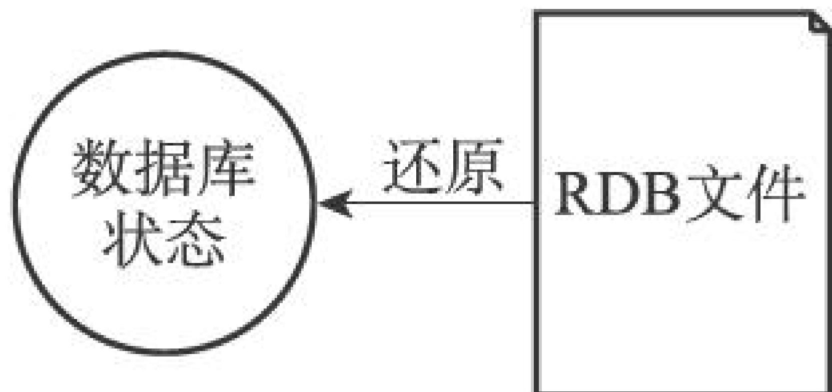


图10-3　用RDB文件来还原数据库状态

因为RDB文件是保存在硬盘里面的，所以即使Redis服务器进程退出，甚至运行Redis服务器的计算机停机，但只要RDB文件仍然存在，Redis服务器就可以用它来还原数据库状态。

本章首先介绍Redis服务器保存和载入RDB文件的方法，重点说明SAVE命令和BGSAVE命令的实现方式。

之后，本章会继续介绍Redis服务器自动保存功能的实现原理。

在介绍完关于保存和载入RDB文件方面的内容之后，我们会详细分析RDB文件中的各个组成部分，并说明这些部分的结构和含义。

在本章的最后，我们将对实际的RDB文件进行分析和解读，将之前学到的关于RDB文件的知识投入到实际应用中。

10.1 RDB文件的创建与载入

有两个Redis命令可以用于生成RDB文件，一个是SAVE，另一个是BGSAVE。

SAVE命令会阻塞Redis服务器进程，直到RDB文件创建完毕为止，在服务器进程阻塞期间，服务器不能处理任何命令请求：

```
redis> SAVE       //
等待直到RDB
文件创建完毕
OK
```

和SAVE命令直接阻塞服务器进程的做法不同，BGSAVE命令会派生出一个子进程，然后由子进程负责创建RDB文件，服务器进程（父进程）继续处理命令请求：

```
redis> BGSAVE     //
派生子进程，并由子进程创建RDB
文件
Background saving started
```

创建RDB文件的实际工作由rdb.c/rdbSave函数完成，SAVE命令和BGSAVE命令会以不同的方式调用这个函数，通过以下伪代码可以明显地看出这两个命令之间的区别：

```
def SAVE():
    #
创建RDB
文件
    rdbSave()
def BGSAVE():
    #
创建子进程
    pid = fork()
    if pid == 0:
        #
子进程负责创建RDB
文件
        rdbSave()
        #
完成之后向父进程发送信号
        signal_parent()
    elif pid > 0:
        #
父进程继续处理命令请求，并通过轮询等待子进程的信号
        handle_request_and_wait_signal()
    else:
        #
处理出错情况
        handle_fork_error()
```

和使用SAVE命令或者BGSAVE命令创建RDB文件不同，RDB文件的载入工作是在服务器启动时自动执行的，所以Redis并没有专门用于载入RDB文件的命令，只要Redis服务器在启动时检测到RDB文件存在，它就

会自动载入RDB文件。

以下是Redis服务器启动时打印的日志记录，其中第二条日志DB loaded from disk:...就是服务器在成功载入RDB文件之后打印的：

```
$ redis-server
[7379] 30 Aug 21:07:01.270 # Server started, Redis version 2.9.11
[7379] 30 Aug 21:07:01.289 * DB loaded from disk: 0.018 seconds
[7379] 30 Aug 21:07:01.289 * The server is now ready to accept connections on port 63
```

另外值得一提的是，因为AOF文件的更新频率通常比RDB文件的更新频率高，所以：

·如果服务器开启了AOF持久化功能，那么服务器会优先使用AOF文件来还原数据库状态。

·只有在AOF持久化功能处于关闭状态时，服务器才会使用RDB文件来还原数据库状态。

服务器判断该用哪个文件来还原数据库状态的流程如图10-4所示。

载入RDB文件的实际工作由rdb.c/rdbLoad函数完成，这个函数和rdbSave函数之间的关系可以用图10-5表示。

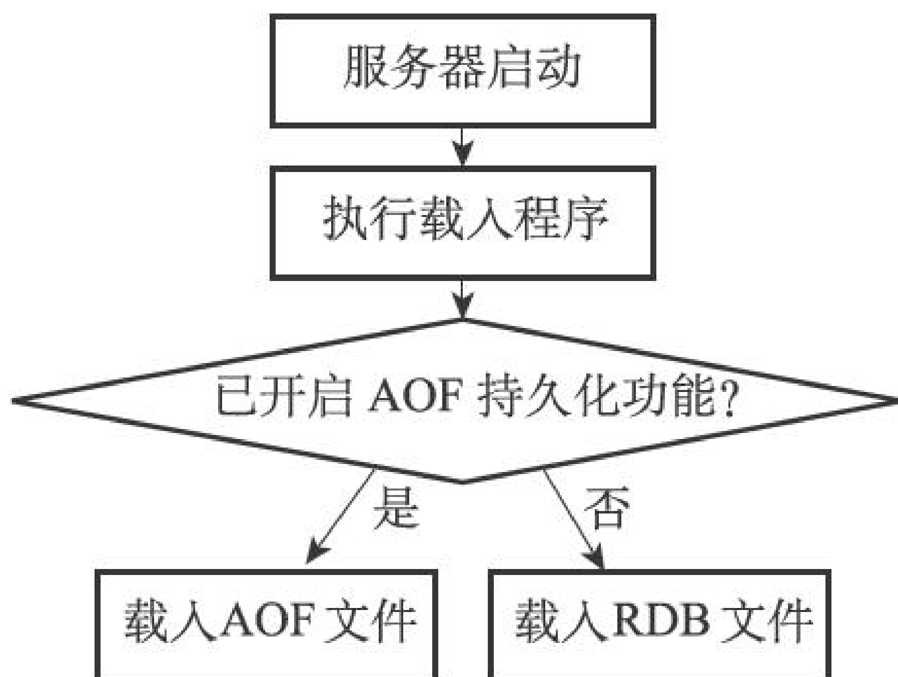


图10-4　服务器载入文件时的判断流程

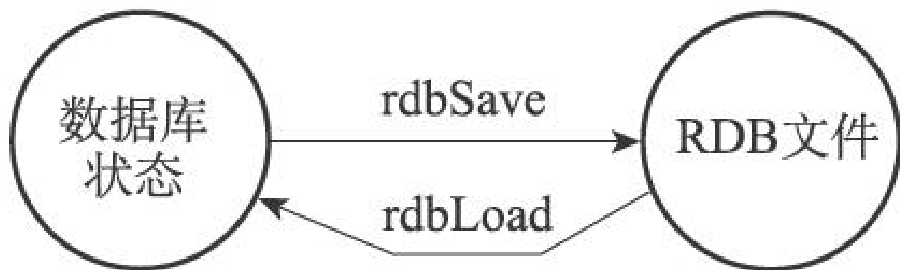


图10-5　创建和载入RDB文件

10.1.1　SAVE命令执行时的服务器状态

前面提到过，当SAVE命令执行时，Redis服务器会被阻塞，所以当SAVE命令正在执行时，客户端发送的所有命令请求都会被拒绝。

只有在服务器执行完SAVE命令、重新开始接受命令请求之后，客户端发送的命令才会被处理。

10.1.2　BGSAVE命令执行时的服务器状态

因为BGSAVE命令的保存工作是由子进程执行的，所以在子进程创建RDB文件的过程中，Redis服务器仍然可以继续处理客户端的命令请求，但是，在BGSAVE命令执行期间，服务器处理SAVE、BGSAVE、BGREWRITEAOF三个命令的方式会和平时有所不同。

首先，在BGSAVE命令执行期间，客户端发送的SAVE命令会被服务器拒绝，服务器禁止SAVE命令和BGSAVE命令同时执行是为了避免父进程（服务器进程）和子进程同时执行两个rdbSave调用，防止产生竞争条件。

其次，在BGSAVE命令执行期间，客户端发送的BGSAVE命令会被服务器拒绝，因为同时执行两个BGSAVE命令也会产生竞争条件。

最后，BGREWRITEAOF和BGSAVE两个命令不能同时执行：

·如果BGSAVE命令正在执行，那么客户端发送的BGREWRITEAOF命令会被延迟到BGSAVE命令执行完毕之后执行。

·如果BGREWRITEAOF命令正在执行，那么客户端发送的BGSAVE命令会被服务器拒绝。

因为BGREWRITEAOF和BGSAVE两个命令的实际工作都由子进程执行，所以这两个命令在操作方面并没有什么冲突的地方，不能同时执行它们只是一个性能方面的考虑——并发出两个子进程，并且这两个子进程都同时执行大量的磁盘写入操作，这怎么想都不会是一个好主意。

10.1.3　RDB文件载入时的服务器状态

服务器在载入RDB文件期间，会一直处于阻塞状态，直到载入工作完成为止。

10.2 自动间隔性保存

在上一节，我们介绍了SAVE命令和BGSAVE的实现方法，并且说明了这两个命令在实现方面的主要区别：SAVE命令由服务器进程执行保存工作，BGSAVE命令则由子进程执行保存工作，所以SAVE命令会阻塞服务器，而BGSAVE命令则不会。

因为BGSAVE命令可以在不阻塞服务器进程的情况下执行，所以Redis允许用户通过设置服务器配置的save选项，让服务器每隔一段时间自动执行一次BGSAVE命令。

用户可以通过save选项设置多个保存条件，但只要其中任意一个条件被满足，服务器就会执行BGSAVE命令。

举个例子，如果我们向服务器提供以下配置：

```
save 900 1
save 300 10
save 60 10000
```

那么只要满足以下三个条件中的任意一个，BGSAVE命令就会被执行：

·服务器在900秒之内，对数据库进行了至少1次修改。

·服务器在300秒之内，对数据库进行了至少10次修改。

·服务器在60秒之内，对数据库进行了至少10000次修改。

举个例子，以下是Redis服务器在60秒之内，对数据库进行了至少10000次修改之后，服务器自动执行BGSAVE命令时打印出来的日志：

```
[5085] 03 Sep 17:09:49.463 * 10000 changes in 60 seconds. Saving...
[5085] 03 Sep 17:09:49.463 * Background saving started by pid 5189
[5189] 03 Sep 17:09:49.522 * DB saved on disk
[5189] 03 Sep 17:09:49.522 * RDB: 0 MB of memory used by copy-on-write
[5085] 03 Sep 17:09:49.563 * Background saving terminated with success
```

在本节接下来的内容中，我们将介绍Redis服务器是如何根据save选项设置的保存条件，自动执行BGSAVE命令的。

10.2.1 设置保存条件

当Redis服务器启动时，用户可以通过指定配置文件或者传入启动参数的方式设置save选项，如果用户没有主动设置save选项，那么服务器会为save选项设置默认条件：

```
save 900 1
save 300 10
save 60 10000
```

接着，服务器程序会根据save选项所设置的保存条件，设置服务器状态redisServer结构的saveparams属性：

```
struct redisServer {
    // ...
    //
记录了保存条件的数组
    struct saveparam *saveparams;
    // ...
};
```

saveparams属性是一个数组，数组中的每个元素都是一个saveparam结构，每个saveparam结构都保存了一个save选项设置的保存条件：

```
struct saveparam {
    //
秒数
    time_t seconds;
    //
修改数
    int changes;
};
```

比如说，如果save选项的值为以下条件：

```
save 900 1
save 300 10
save 60 10000
```

那么服务器状态中的saveparams数组将会是图10-6所示的样子。

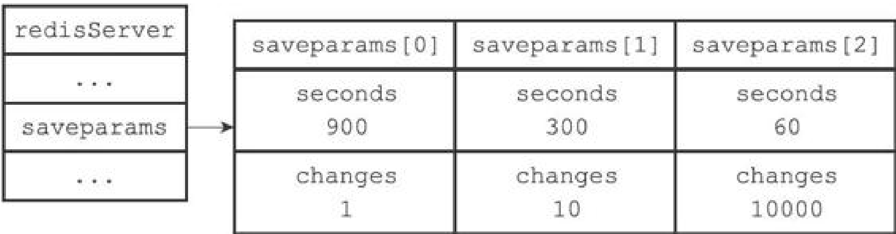


图10-6　服务器状态中的保存条件

10.2.2　dirty计数器和lastsave属性

除了saveparams数组之外，服务器状态还维持着一个dirty计数器，以及一个lastsave属性：

·dirty计数器记录距离上一次成功执行SAVE命令或者BGSAVE命令

之后，服务器对数据库状态（服务器中的所有数据库）进行了多少次修改（包括写入、删除、更新等操作）。

·lastsave属性是一个UNIX时间戳，记录了服务器上一次成功执行SAVE命令或者BGSAVE命令的时间。

```
struct redisServer {
    // ...
    //
修改计数器
    long long dirty;
    //
上一次执行保存的时间
    time_t lastsave;
    // ...
};
```

当服务器成功执行一个数据库修改命令之后，程序就会对dirty计数器进行更新：命令修改了多少次数据库，dirty计数器的值就增加多少。

例如，如果我们为一个字符串键设置值：

```
redis> SET message "hello"
OK
```

那么程序会将dirty计数器的值增加1。

又例如，如果我们向一个集合键增加三个新元素：

```
redis> SADD database Redis MongoDB MariaDB
(integer) 3
```

那么程序会将dirty计数器的值增加3。



图10-7　服务器状态示例

图10-7展示了服务器状态中包含的dirty计数器和lastsave属性，说明如下：

·dirty计数器的值为123，表示服务器在上次保存之后，对数据库状态共进行了123次修改。

·lastsave属性则记录了服务器上次执行保存操作的时间1378270800（2013年9月4日零时）。

10.2.3　检查保存条件是否满足

Redis的服务器周期性操作函数serverCron默认每隔100毫秒就会执行一次，该函数用于对正在运行的服务器进行维护，它的其中一项工作就是检查save选项所设置的保存条件是否已经满足，如果满足的话，就执行BGSAVE命令。

以下伪代码展示了serverCron函数检查保存条件的过程：

```
def serverCron():
    # ...
    #
遍历所有保存条件
    for saveparam in server.saveparams:
        #
计算距离上次执行保存操作有多少秒
        save_interval = unixtime_now()-server.lastsave
        #
如果数据库状态的修改次数超过条件所设置的次数
        #
```

```
            # 并且距离上次保存的时间超过条件所设置的时间
            # 那么执行保存操作
            if      server.dirty >= saveparam.changes and \
               save_interval > saveparam.seconds:
                BGSAVE()
    # ...
```

程序会遍历并检查saveparams数组中的所有保存条件，只要有任意一个条件被满足，那么服务器就会执行BGSAVE命令。

举个例子，如果Redis服务器的当前状态如图10-8所示。

redisServer
...
saveparams
...
dirty 123
lastsave 1378270800
...

saveparams[0]	saveparams[1]	saveparams[2]
seconds 900	seconds 300	seconds 60
changes 1	changes 10	changes 10000

图10-8　服务器状态

那么当时间来到1378271101，也即是1378270800的301秒之后，服务器将自动执行一次BGSAVE命令，因为saveparams数组的第二个保存条件——300秒之内有至少10次修改——已经被满足。

假设BGSAVE在执行5秒之后完成，那么图10-8所示的服务器状态将更新为图10-9，其中dirty计数器已经被重置为0，而lastsave属性也被更新为1378271106。

redisServer
...
saveparams
...
dirty 0
lastsave 1378271106
...

saveparams[0]	saveparams[1]	saveparams[2]
seconds 900	seconds 300	seconds 60
changes 1	changes 10	changes 10000

图10-9　执行BGSAVE之后的服务器状态

以上就是Redis服务器根据save选项所设置的保存条件，自动执行BGSAVE命令，进行间隔性数据保存的实现原理。

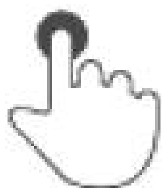
10.3　RDB文件结构

在本章之前的内容中，我们介绍了Redis服务器保存和载入RDB文件的方法，在这一节，我们将对RDB文件本身进行介绍，并详细说明文件各个部分的结构和意义。

图10-10展示了一个完整RDB文件所包含的各个部分。

REDIS	db_version	databases	EOF	check_sum

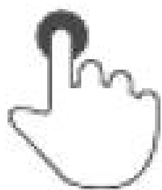
图10-10　RDB文件结构



注意

为了方便区分变量、数据、常量，图10-10中用全大写单词标示常量，用全小写单词标示变量和数据。本章展示的所有RDB文件结构图都遵循这一规则。

RDB文件的最开头是REDIS部分，这个部分的长度为5字节，保存着“REDIS”五个字符。通过这五个字符，程序可以在载入文件时，快速检查所载入的文件是否RDB文件。



注意

因为RDB文件保存的是二进制数据，而不是C字符串，为了简便起见，我们用"REDIS"符号代表'R'、'E'、'D'、'I'、'S'五个字符，而不是带'\0'结尾符号的C字符串'R'、'E'、'D'、'I'、'S'、'\0'。本章介绍的所有内容，以及展示的所有RDB文件结构图都遵循这一规则。

db_version长度为4字节，它的值是一个字符串表示的整数，这个整数记录了RDB文件的版本号，比如"0006"就代表RDB文件的版本为第六版。本章只介绍第六版RDB文件的结构。

databases部分包含着零个或任意多个数据库，以及各个数据库中的键值对数据：

·如果服务器的数据库状态为空（所有数据库都是空的），那么这个部分也为空，长度为0字节。

·如果服务器的数据库状态为非空（有至少一个数据库非空），那么这个部分也为非空，根据数据库所保存键值对的数量、类型和内容不同，这个部分的长度也会有所不同。

EOF常量的长度为1字节，这个常量标志着RDB文件正文内容的结束，当读入程序遇到这个值的时候，它知道所有数据库的所有键值对都已经载入完毕了。

check_sum是一个8字节长的无符号整数，保存着一个校验和，这个校验和是程序通过对REDIS、db_version、databases、EOF四个部分的内容进行计算得出的。服务器在载入RDB文件时，会将载入数据所计算出的校验和与check_sum所记录的校验和进行对比，以此来检查RDB文件是否有出错或者损坏的情况出现。

作为例子，图10-11展示了一个databases部分为空的RDB文件：文件开头的"REDIS"表示这是一个RDB文件，之后的"0006"表示这是第六版的RDB文件，因为databases为空，所以版本号之后直接跟着EOF常量，最后的6265312314761917404是文件的校验和。

"REDIS"	"0006"	EOF	6265312314761917404

图10-11　databases部分为空的RDB文件

10.3.1　databases部分

一个RDB文件的databases部分可以保存任意多个非空数据库。

例如，如果服务器的0号数据库和3号数据库非空，那么服务器将创建一个如图10-12所示的RDB文件，图中的database　0代表0号数据库中的所有键值对数据，而database　3则代表3号数据库中的所有键值对数据。

REDIS	db_version	database 0	database 3	EOF	check_sum

图10-12　带有两个非空数据库的RDB文件示例

每个非空数据库在RDB文件中都可以保存为SELECTDB、db_number、key_value_pairs三个部分，如图10-13所示。

SELECTDB	db_number	key_value_pairs

图10-13　RDB文件中的数据库结构

SELECTDB常量的长度为1字节，当读入程序遇到这个值的时候，它知道接下来要读入的将是一个数据库号码。

db_number保存着一个数据库号码，根据号码的大小不同，这个部分的长度可以是1字节、2字节或者5字节。当程序读入db_number部分之后，服务器会调用SELECT命令，根据读入的数据库号码进行数据库切换，使得之后读入的键值对可以载入到正确的数据库中。

key_value_pairs部分保存了数据库中的所有键值对数据，如果键值对带有过期时间，那么过期时间也会和键值对保存在一起。根据键值对的数量、类型、内容以及是否有过期时间等条件的不同，key_value_pairs部分的长度也会有所不同。

作为例子，图10-14展示了RDB文件中，0号数据库的结构。



图10-14　数据库结构示例

另外，图10-15则展示了一个完整的RDB文件，文件中包含了0号数据库和3号数据库。



图10-15　RDB文件中的数据库结构示例

10.3.2　key_value_pairs部分

RDB文件中的每个key_value_pairs部分都保存了一个或以上数量的键值对，如果键值对带有过期时间的话，那么键值对的过期时间也会被保存在内。

不带过期时间的键值对在RDB文件中由TYPE、key、value三部分组成，如图10-16所示。



图10-16　不带过期时间的键值对

TYPE记录了value的类型，长度为1字节，值可以是以下常量的其中一个：

·REDIS_RDB_TYPE_STRING

·REDIS_RDB_TYPE_LIST

·REDIS_RDB_TYPE_SET

·REDIS_RDB_TYPE_ZSET

·REDIS_RDB_TYPE_HASH

·REDIS_RDB_TYPE_LIST_ZIPLIST

·REDIS_RDB_TYPE_SET_INTSET

·REDIS_RDB_TYPE_ZSET_ZIPLIST

·REDIS_RDB_TYPE_HASH_ZIPLIST

以上列出的每个TYPE常量都代表了一种对象类型或者底层编码，当服务器读入RDB文件中的键值对数据时，程序会根据TYPE的值来决定如何读入和解释value的数据。key和value分别保存了键值对的键对象和值对象：

·其中key总是一个字符串对象，它的编码方式和REDIS_RDB_TYPE_STRING类型的value一样。根据内容长度的不同，key的长度也会有所不同。

·根据TYPE类型的不同，以及保存内容长度的不同，保存value的结构和长度也会有所不同，本节稍后会详细说明每种TYPE类型的value结构保存方式。

带有过期时间的键值对在RDB文件中的结构如图10-17所示。

EXPIRETIME_MS	ms	TYPE	key	value

图10-17　带有过期时间的键值对

带有过期时间的键值对中的TYPE、key、value三个部分的意义，和前面介绍的不带过期时间的键值对的TYPE、key、value三个部分的意义完全相同，至于新增的EXPIRETIME_MS和ms，它们的意义如下：

·EXPIRETIME_MS常量的长度为1字节，它告知读入程序，接下来要读入的将是一个以毫秒为单位的过期时间。

·ms是一个8字节长的带符号整数，记录着一个以毫秒为单位的UNIX时间戳，这个时间戳就是键值对的过期时间。

REDIS_RDB_TYPE_STRING	key	value

图10-18 无过期时间的字符串键值对示例

作为例子，图10-18展示了一个没有过期时间的字符串键值对。

图10-19展示了一个带有过期时间的集合键值对，其中键的过期时间为1388556000000（2014年1月1日零时）。

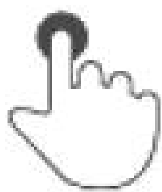
EXPIRETIME_MS	1388556000000	REDIS_RDB_TYPE_SET	key	value

图10-19　带有过期时间的集合键值对示例

10.3.3　value的编码

RDB文件中的每个value部分都保存了一个值对象，每个值对象的类型都由与之对应的TYPE记录，根据类型的不同，value部分的结构、长度也会有所不同。

在接下来的各个小节中，我们将分别介绍各种不同类型的值对象在RDB文件中的保存结构。


注意

本节接下来说到的各种REDIS_ENCODING_*编码曾经在第8章中介绍过，如果忘记了可以去回顾一下。

1.字符串对象

如果TYPE的值为REDIS_RDB_TYPE_STRING，那么value保存的就是一个字符串对象，字符串对象的编码可以是REDIS_ENCODING_INT或者REDIS_ENCODING_RAW。

如果字符串对象的编码为REDIS_ENCODING_INT，那么说明对象中保存的是长度不超过32位的整数，这种编码的对象将以图10-20所示的结构保存。

其中，ENCODING的值可以是REDIS_RDB_ENC_INT8、REDIS_RDB_ENC_INT16或者REDIS_RDB_ENC_INT32三个常量的其中一个，它们分别代表RDB文件使用8位（bit）、16位或者32位来保存整数值integer。

举个例子，如果字符串对象中保存的是可以用8位来保存的整数123，那么这个对象在RDB文件中保存的结构将如图10-21所示。



图10-20　INT编码字符串对象的保存结构

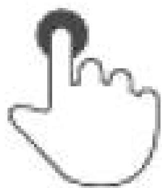


图10-21　用8位来保存整数的例子

如果字符串对象的编码为REDIS_ENCODING_RAW，那么说明对象所保存的是一个字符串值，根据字符串长度的不同，有压缩和不压缩两种方法来保存这个字符串：

·如果字符串的长度小于等于20字节，那么这个字符串会直接被原样保存。

·如果字符串的长度大于20字节，那么这个字符串会被压缩之后再保存。



注意

以上两个条件是在假设服务器打开了RDB文件压缩功能的情况下进行的，如果服务器关闭了RDB文件压缩功能，那么RDB程序总以无压缩的方式保存字符串值。

具体信息可以参考redis.conf文件中关于rdbcompression选项的说明。

对于没有被压缩的字符串，RDB程序会以图10-22所示的结构来保存该字符串。



图10-22　无压缩字符串的保存结构

其中，string部分保存了字符串值本身，而len保存了字符串值的长度。对于压缩后的字符串，RDB程序会以图10-23所示的结构来保存该字符串。

REDIS_RDB_ENC_LZF	compressed_len	origin_len	compressed_string

图10-23　压缩后字符串的保存结构

其中，REDIS_RDB_ENC_LZF常量标志着字符串已经被LZF算法（http://liblzf.plan9.de）压缩过了，读入程序在碰到这个常量时，会根据之后的compressed_len、origin_len和compressed_string三部分，对字符串进行解压缩：其中compressed_len记录的是字符串被压缩之后的长度，而origin_len记录的是字符串原来的长度，compressed_string记录的则是被压缩之后的字符串。

图10-24展示了一个保存无压缩字符串的例子，其中字符串的长度为5，字符串的值为"hello"。

图10-25展示了一个压缩后的字符串示例，从图中可以看出，字符串原本的长度为21，压缩之后的长度为6，压缩之后的字符串内容为"?aa???"，其中?代表的是无法用字符串形式打印出来的字节。

5	"hello"

图10-24　无压缩的字符串

REDIS_RDB_ENC_LZF	6	21	"?aa???"

图10-25　压缩后的字符串

2.列表对象

如果TYPE的值为REDIS_RDB_TYPE_LIST，那么value保存的就是一个REDIS_ENCODING_LINKEDLIST编码的列表对象，RDB文件保存这种对象的结构如图10-26所示。

list_length	item1	item2	...	itemN

图10-26　LINKEDLIST编码列表对象的保存结构

list_length记录了列表的长度，它记录列表保存了多少个项（item），读入程序可以通过这个长度知道自己应该读入多少个列表项。

图中以item开头的部分代表列表的项，因为每个列表项都是一个字

符串对象，所以程序会以处理字符串对象的方式来保存和读入列表项。

作为示例，图10-27展示了一个包含三个元素的列表。

3	5	"hello"	5	"world"	1	"!"

图10-27　保存LINKEDLIST编码列表的例子

结构中的第一个数字3是列表的长度，之后跟着的分别是第一个列表项、第二个列表项和第三个列表项，其中：

·第一个列表项的长度为5，内容为字符串"hello"。

·第二个列表项的长度也为5，内容为字符串"world"。

·第三个列表项的长度为1，内容为字符串"！"。

3.集合对象

如果TYPE的值为REDIS_RDB_TYPE_SET，那么value保存的就是一个REDIS_ENCODING_HT编码的集合对象，RDB文件保存这种对象的结构如图10-28所示。

set_size	elem1	elem2	...	elemN

图10-28　HT编码集合对象的保存结构

其中，set_size是集合的大小，它记录集合保存了多少个元素，读入程序可以通过这个大小知道自己应该读入多少个集合元素。

图中以elem开头的部分代表集合的元素，因为每个集合元素都是一个字符串对象，所以程序会以处理字符串对象的方式来保存和读入集合元素。

作为示例，图10-29展示了一个包含四个元素的集合。

4	5	"apple"	6	"banana"	3	"cat"	3	"dog"

图10-29　保存HT编码集合的例子

结构中的第一个数字4记录了集合的大小，之后跟着的是集合的四个元素：

·第一个元素的长度为5，值为"apple"。

·第二个元素的长度为6，值为"banana"。

·第三个元素的长度为3，值为"cat"。

·第四个元素的长度为3，值为"dog"。

4.哈希表对象

如果TYPE的值为REDIS_RDB_TYPE_HASH，那么value保存的就是一个REDIS_ENCODING_HT编码的集合对象，RDB文件保存这种对象的结构如图10-30所示：

·hash_size记录了哈希表的大小，也即是这个哈希表保存了多少键值对，读入程序可以通过这个大小知道自己应该读入多少个键值对。

·以key_value_pair开头的部分代表哈希表中的键值对，键值对的键和值都是字符串对象，所以程序会以处理字符串对象的方式来保存和读入键值对。

hash_size	key_value_pair 1	key_value_pair 2	...	key_value_pair N

图10-30　HT编码哈希表对象的保存结构

结构中的每个键值对都以键紧挨着值的方式排列在一起，如图10-31所示。

key1	value1	key2	value2	key3	value3	...

图10-31　键值对的保存结构

因此，从更详细的角度看，图10-30所展示的结构可以进一步修改为图10-32。

hash_size	key1	value1	key2	value2	...	keyN	valueN

图10-32　更详细的HT编码哈希表对象的保存结构

作为示例，图10-33展示了一个包含两个键值对的哈希表。

2	1	"a"	5	"apple"	1	"b"	6	"banana"

图10-33　保存HT编码哈希表的例子

在这个示例结构中，第一个数字2记录了哈希表的键值对数量，之后跟着的是两个键值对：

·第一个键值对的键是长度为1的字符串"a"，值是长度为5的字符串"apple"。

·第二个键值对的键是长度为1的字符串"b"，值是长度为6的字符

串"banana"。

5.有序集合对象

如果TYPE的值为REDIS_RDB_TYPE_ZSET，那么value保存的就是一个REDIS_ENCODING_SKIPLIST编码的有序集合对象，RDB文件保存这种对象的结构如图10-34所示。

sorted_set_size	element1	element2	...	elementN

图10-34　SKIPLIST编码有序集合对象的保存结构

sorted_set_size记录了有序集合的大小，也即是这个有序集合保存了多少元素，读入程序需要根据这个值来决定应该读入多少有序集合元素。

以element开头的部分代表有序集合中的元素，每个元素又分为成员（member）和分值（score）两部分，成员是一个字符串对象，分值则是一个double类型的浮点数，程序在保存RDB文件时会先将分值转换成字符串对象，然后再用保存字符串对象的方法将分值保存起来。

有序集合中的每个元素都以成员紧挨着分值的方式排列，如图10-35所示。

member1	score1	member2	score2	member3	score3	...

图10-35　成员和分值的保存结构

因此，从更详细的角度看，图10-34所展示的结构可以进一步修改为图10-36。

sorted_set_size	member1	score1	member2	score2	...	memberN	scoreN

图10-36　更详细的SKIPLIST编码有序集合对象的保存结构

作为示例，图10-37展示了一个带有两个元素的有序集合。

2	2	"pi"	4	"3.14"	1	"e"	3	"2.7"

图10-37　保存SKIPLIST编码有序集合的例子

在这个示例结构中，第一个数字2记录了有序集合的元素数量，之后跟着的是两个有序集合元素：

·第一个元素的成员是长度为2的字符串"pi"，分值被转换成字符串之后变成了长度为4的字符串"3.14"。

·第二个元素的成员是长度为1的字符串"e"，分值被转换成字符串之

后变成了长度为3的字符串"2.7"。

6.INTSET编码的集合

如果TYPE的值为REDIS_RDB_TYPE_SET_INTSET，那么value保存的就是一个整数集合对象，RDB文件保存这种对象的方法是，先将整数集合转换为字符串对象，然后将这个字符串对象保存到RDB文件里面。

如果程序在读入RDB文件的过程中，碰到由整数集合对象转换成的字符串对象，那么程序会根据TYPE值的指示，先读入字符串对象，再将这个字符串对象转换成原来的整数集合对象。

7.ZIPLIST编码的列表、哈希表或者有序集合

如果TYPE的值为REDIS_RDB_TYPE_LIST_ZIPLIST、REDIS_RDB_TYPE_HASH_ZIPLIST或者REDIS_RDB_TYPE_ZSET_ZIPLIST，那么value保存的就是一个压缩列表对象，RDB文件保存这种对象的方法是：

1）将压缩列表转换成一个字符串对象。

2）将转换所得的字符串对象保存到RDB文件。

如果程序在读入RDB文件的过程中，碰到由压缩列表对象转换成的字符串对象，那么程序会根据TYPE值的指示，执行以下操作：

1）读入字符串对象，并将它转换成原来的压缩列表对象。

2）根据TYPE的值，设置压缩列表对象的类型：如果TYPE的值为REDIS_RDB_TYPE_LIST_ZIPLIST，那么压缩列表对象的类型为列表；如果TYPE的值为REDIS_RDB_TYPE_HASH_ZIPLIST，那么压缩列表对象的类型为哈希表；如果TYPE的值为REDIS_RDB_TYPE_ZSET_ZIPLIST，那么压缩列表对象的类型为有序集合。

从步骤2可以看出，由于TYPE的存在，即使列表、哈希表和有序集合三种类型都使用压缩列表来保存，RDB读入程序也总可以将读入并转换之后得出的压缩列表设置成原来的类型。

10.4 分析RDB文件

通过上一节对RDB文件的介绍，我们现在应该对RDB文件中的各种内容和结构有一定的了解了，是时候抛开单纯的图片示例，开始分析和观察一下实际的RDB文件了。

我们使用od命令来分析Redis服务器产生的RDB文件，该命令可以用给定的格式转存（dump）并打印输入文件。比如说，给定-c参数可以以ASCII编码的方式打印输入文件，给定-x参数可以以十六进制的方式打印输入文件，诸如此类，具体的信息可以参考od命令的文档。

10.4.1 不包含任何键值对的RDB文件

让我们首先从最简单的情况开始，执行以下命令，创建一个数据库状态为空的RDB文件：

```
redis> FLUSHALL
OK
redis> SAVE
OK
```

然后调用od命令，打印RDB文件：

```
$ od -c dump.rdb
0000000   R  E  D  I  S  0  0  0  6 377 334 263  C 360  Z 334
0000020 362  V
0000022
```

根据之前学习的RDB文件结构知识，当一个RDB文件没有包含任何数据库数据时，这个RDB文件将由以下四个部分组成：

·五个字节的"REDIS"字符串。

·四个字节的版本号（db_version）。

·一个字节的EOF常量。

·八个字节的校验和（check_sum）。

从od命令的输出中可以看到，最开头的是“REDIS”字符串，之后的0006是版本号，再之后的一个字节377代表EOF常量，最后的334 263 C 360 Z 334 362 V八个字节则代表RDB文件的校验和。

10.4.2 包含字符串键的RDB文件

这次我们来分析一个带有单个字符串键的数据库：

```
redis> FLUSHALL
OK
redis> SET MSG "HELLO"
OK
redis> SAVE
OK
```

再次执行od命令：

```
$ od -c dump.rdb
0000000   R   E   D   I   S   0   0   0   6 376  \0 \0 003   M   S   G
0000020 005   H   E   L   L   O 377 207   z   =  304   f   T   L 343
0000037
```

根据之前学习的数据库结构知识，当一个数据库被保存到RDB文件时，这个数据库将由以下三部分组成：

·一个一字节长的特殊值SELECTDB。

·一个长度可能为一字节、两字节或者五字节的数据库号码（db_number）。

·一个或以上数量的键值对（key_value_pairs）。

观察od命令打印的输出，RDB文件的最开始仍然是REDIS和版本号0006，之后出现的376代表SELECTDB常量，再之后的\0代表整数0，表示被保存的数据库为0号数据库。

在数据库号码之后，直到代表EOF常量的377为止，RDB文件包含有以下内容：

```
\0 003 M S G 005 H E L L O
```

根据之前学习的键值对结构知识，在RDB文件中，没有过期时间的键值对由类型（TYPE）、键（key）、值（value）三部分组成：其中类型的长度为一字节，键和值都是字符串对象，并且字符串在未被压缩前，都是以字符串长度为前缀，后跟字符串内容本身的方式来储存的。

根据这些特征，我们可以确定\0就是字符串类型的TYPE值REDIS_RDB_TYPE_STRING（这个常量的实际值为整数0），之后的003是键MSG的长度值，再之后的005则是值HELLO的长度。

10.4.3　包含带有过期时间的字符串键的RDB文件

现在，让我们来创建一个带有过期时间的字符串键：

```
redis> FLUSHALL
OK
redis> SETEX MSG 10086 "HELLO"
```

```
OK
redis> SAVE
OK
```

打印RDB文件：

```
$ od -c dump.rdb
0000000   R   E   D   I   S   0   0   0   6 376 \0 374   \   2 365 336
0000020   @ 001 \0  \0  \0 003   M   S   G 005   H   E   L   L   O 377
0000040 212 231   x 247 252   } 021 306
0000050
```

根据之前学习的键值对结构知识，一个带有过期时间的键值对将由以下部分组成：

·一个一字节长的EXPIRETIME_MS特殊值。

·一个八字节长的过期时间（ms）。

·一个一字节长的类型（TYPE）。

·一个键（key）和一个值（value）。

根据这些特征，可以得出RDB文件各个部分的意义：

·REDIS0006：RDB文件标志和版本号。

·376\0：切换到0号数据库。

·374：代表特殊值EXPIRETIME_MS。

·\2 365 336@001\0\0：代表八字节长的过期时间。

·\0 003 M S G：\0表示这是一个字符串键，003是键的长度，MSG是键。

·005 H E L L O：005是值的长度，HELLO是值。

·377：代表EOF常量。

·212 231 x 247 252 } 021 306：代表八字节长的校验和。

10.4.4 包含一个集合键的RDB文件

最后，让我们试试在RDB文件中包含集合键：

```
redis> FLUSHALL
OK
redis> SADD LANG "C" "JAVA" "RUBY"
(integer) 3
redis> SAVE
```

```
OK
```

打印输出如下：

```
$ od -c dump.rdb
0000000   R   E   D   I   S   0   0   0   6 376 \0 002 004 L   A   N
0000020   G 003 004   R   U   B   Y 004 J   A   V   A 001 C 377 202
0000040 312   r 352 346 305   * 023
0000047
```

以下是RDB文件各个部分的意义：

·REDIS0006：RDB文件标志和版本号。

·376\0：切换到0号数据库。

·002 004 L A N G：002是常量REDIS_RDB_TYPE_SET（这个常量的实际值为整数2），表示这是一个哈希表编码的集合键，004表示键的长度，LANG是键的名字。

·003：集合的大小，说明这个集合包含三个元素。

·004 R U B Y：集合的第一个元素。

·004 J A V A：集合的第二个元素。

·001 C：集合的第三个元素。

·377：代表常量EOF。

·202 312 r 352 346 305*023：代表校验和。

10.4.5　关于分析RDB文件的说明

因为Redis本身带有RDB文件检查工具redis-check-dump，网上也能找到很多处理RDB文件的工具，所以人工分析RDB文件的内容并不是学习Redis所必须掌握的技能。

不过从学习RDB文件的角度来看，人工分析RDB文件是一个不错的练习，这种练习可以帮助我们熟悉RDB文件的结构和格式，如果读者有兴趣的话，可以在理解本章的内容之后，适当地尝试一下。

最后要提醒的是，前面我们一直用od命令配合-c参数来打印RDB文件，因为使用ASCII编码打印RDB文件可以很容易地发现文件中的字符串内容。

但是，对于RDB文件中的数字值，比如校验和来说，通过ASCII编码来打印它并不容易看出它的真实值，更好的办法是使用-cx参数调用od命

令，同时以ASCII编码和十六进制格式打印RDB文件：

```
$ od -cx dump.rdb
0000000   R   E  D   I  S   0  0   0   6 377   334 263   C  360   Z  334
          4552  4944  3053  3030    ff36      b3dc    f043    dc5a
0000020  362   V
          56f2
0000022
```

现在可以从输出中看出，RDB文件的校验和为0x 56f2 dc5a f043 b3dc（校验和以小端方式保存），这比用ASCII编码打印出来的334 263 C360 Z 334 362 V要清晰得多，后者看起来就像乱码一样。

10.5 重点回顾

·RDB文件用于保存和还原Redis服务器所有数据库中的所有键值对数据。

·SAVE命令由服务器进程直接执行保存操作，所以该命令会阻塞服务器。

·BGSAVE令由子进程执行保存操作，所以该命令不会阻塞服务器。

·服务器状态中会保存所有用save选项设置的保存条件，当任意一个保存条件被满足时，服务器会自动执行BGSAVE命令。

·RDB文件是一个经过压缩的二进制文件，由多个部分组成。

·对于不同类型的键值对，RDB文件会使用不同的方式来保存它们。

10.6 参考资料

·Sripathi Krishnan编写的《Redis RDB文件格式》文档以文字的形式详细记录了RDB文件的格式，如果想深入理解RDB文件，或者为RDB文件编写分析/载入程序，那么这篇文档会是很好的参考资料：https://github.com/sripathikrishnan/redis-rdb-tools/wiki/Redis-RDB-Dump-File-Format。

·Sripathi Krishnan编写的《Redis RDB版本历史》也详细地记录了RDB文件在各个版本中的变化，因为本章只介绍了Redis 2.6或以上版本目前正在使用的第六版RDB文件，而没有对其他版本的RDB文件进行介绍，所以如果读者对RDB文件的演进历史感兴趣，或者要处理不同版本的RDB文件的话，那么这篇文档会是很好的资料：https://github.com/sripathikrishnan/redis-rdbtools/blob/master/docs/RDB_Version_History.textile。

·Redis作者的博文《Redis persistence demystified》很好地解释了Redis的持久化功能和其他常见数据库的持久化功能之间的异同，非常值得一读：http://oldblog.antirez.com/post/redispersistence-demystified.html，NoSQLFan网站上有这篇文章的翻译版《解密Redis持久化》：http://blog.nosqlfan.com/html/3813.html。

第11章　AOF持久化

除了RDB持久化功能之外，Redis还提供了AOF（Append Only File）持久化功能。与RDB持久化通过保存数据库中的键值对来记录数据库状态不同，AOF持久化是通过保存Redis服务器所执行的写命令来记录数据库状态的，如图11-1所示。



图11-1　AOF持久化

举个例子，如果我们对空白的数据库执行以下写命令，那么数据库中将包含三个键值对：

```
redis> SET msg "hello"
OK
redis> SADD fruits "apple" "banana" "cherry"
(integer) 3
redis> RPUSH numbers 128 256 512
(integer) 3
```

RDB持久化保存数据库状态的方法是将msg、fruits、numbers三个键的键值对保存到RDB文件中，而AOF持久化保存数据库状态的方法则是将服务器执行的SET、SADD、RPUSH三个命令保存到AOF文件中。

被写入AOF文件的所有命令都是以Redis的命令请求协议格式保存的，因为Redis的命令请求协议是纯文本格式，所以我们可以直接打开一个AOF文件，观察里面的内容。

例如，对于之前执行的三个写命令来说，服务器将产生包含以下内容的AOF文件：

```
*2\r\n$6\r\nSELECT\r\n$1\r\n0\r\n
*3\r\n$3\r\nSET\r\n$3\r\nmsg\r\n$5\r\nhello\r\n
*5\r\n$4\r\nSADD\r\n$6\r\nfruits\r\n$5\r\napple\r\n$6\r\nbanana\r\n$6\r
\ncherry\r\n
*5\r\n$5\r\nRPUSH\r\n$7\r\nnumbers\r\n$3\r\n128\r\n$3\r\n256\r\n$3\r\n512\r\n
```

在这个AOF文件里面，除了用于指定数据库的SELECT命令是服务器自动添加的之外，其他都是我们之前通过客户端发送的命令。

服务器在启动时，可以通过载入和执行AOF文件中保存的命令来还原服务器关闭之前的数据库状态，以下就是服务器载入AOF文件并还原数据库状态时打印的日志：

```
[8321] 05 Sep 11:58:50.448 # Server started, Redisversion 2.9.11
[8321] 05 Sep 11:58:50.449 * DB loaded from append only file: 0.000 seconds
[8321] 05 Sep 11:58:50.449 * The server is now ready to accept connections on port 63
```

在本章接下来的内容中，我们将对AOF持久化功能的实现进行介绍，说明AOF文件的写入、保存、载入等操作的实现原理。

之后我们还会介绍用于减少AOF文件体积的AOF重写功能，以及该功能的实现原理。

11.1 AOF持久化的实现

AOF持久化功能的实现可以分为命令追加（append）、文件写入、文件同步（sync）三个步骤。

11.1.1 命令追加

当AOF持久化功能处于打开状态时，服务器在执行完一个写命令之后，会以协议格式将被执行的写命令追加到服务器状态的aof_buf缓冲区的末尾：

```
struct redisServer {
    // ...
    // AOF
缓冲区
    sds aof_buf;
    // ...
};
```

举个例子，如果客户端向服务器发送以下命令：

```
redis> SET KEY VALUE
OK
```

那么服务器在执行这个SET命令之后，会将以下协议内容追加到aof_buf缓冲区的末尾：

```
*3\r\n$3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n
```

又例如，如果客户端向服务器发送以下命令：

```
redis> RPUSH NUMBERS ONE TWO THREE
(integer) 3
```

那么服务器在执行这个RPUSH命令之后，会将以下协议内容追加到aof_buf缓冲区的末尾：

```
*5\r\n$5\r\nRPUSH\r\n$7\r\nNUMBERS\r\n$3\r\nONE\r\n$3\r\nTWO\r\n$5\r\nTHREE\r
\n
```

以上就是AOF持久化的命令追加步骤的实现原理。

11.1.2 AOF文件的写入与同步

Redis的服务器进程就是一个事件循环（loop），这个循环中的文件事件负责接收客户端的命令请求，以及向客户端发送命令回复，而时间事件则负责执行像serverCron函数这样需要定时运行的函数。

因为服务器在处理文件事件时可能会执行写命令，使得一些内容被追加到aof_buf缓冲区里面，所以在服务器每次结束一个事件循环之前，它都会调用flushAppendOnlyFile函数，考虑是否需要将aof_buf缓冲区中的内容写入和保存到AOF文件里面，这个过程可以用以下伪代码表示：

```
def eventLoop():
    while True:
        #
处理文件事件，接收命令请求以及发送命令回复
        #
处理命令请求时可能会有新内容被追加到 aof_buf
缓冲区中
        processFileEvents()
        #
处理时间事件
        processTimeEvents()
        #
考虑是否要将 aof_buf
中的内容写入和保存到 AOF
文件里面
        flushAppendOnlyFile()
```

flushAppendOnlyFile函数的行为由服务器配置的appendfsync选项的值来决定，各个不同值产生的行为如表11-1所示。

表11-1　不同appendfsync值产生不同的持久化行为

appendfsync 选项的值	flushAppendOnlyFile 函数的行为
always	将 aof_buf 缓冲区中的所有内容写入并同步到 AOF 文件
everysec	将 aof_buf 缓冲区中的所有内容写入到 AOF 文件，如果上次同步 AOF 文件的时间距离现在超过一秒钟，那么再次对 AOF 文件进行同步，并且这个同步操作是由一个线程专门负责执行的
no	将 aof_buf 缓冲区中的所有内容写入到 AOF 文件，但并不对 AOF 文件进行同步，何时同步由操作系统来决定

如果用户没有主动为appendfsync选项设置值，那么appendfsync选项的默认值为everysec，关于appendfsync选项的更多信息，请参考Redis项目附带的示例配置文件redis.conf。

文件的写入和同步

为了提高文件的写入效率，在现代操作系统中，当用户调用write函数，将一些数据写入到文件的时候，操作系统通常会将写入数据暂时保存在一个内存缓冲区里面，等到缓冲区的空间被填满、或者超过了指定的时限之后，才真正地将缓冲区中的数据写入到磁盘里面。

这种做法虽然提高了效率，但也为写入数据带来了安全问题，

因为如果计算机发生停机，那么保存在内存缓冲区里面的写入数据将会丢失。

为此，系统提供了fsync和fdatasync两个同步函数，它们可以强制让操作系统立即将缓冲区中的数据写入到硬盘里面，从而确保写入数据的安全性。

举个例子，假设服务器在处理文件事件期间，执行了以下三个写入命令：

```
1) SADD databases "Redis" "MongoDB" "MariaDB"
2) SET date "2013-9-5"
3) INCR click_counter 10086
```

那么aof_buf缓冲区将包含这三个命令的协议内容：

```
*5\r\n$4\r\nSADD\r\n$9\r\ndatabases\r\n$5\r\nRedis\r\n$7\r\nMongoDB\r\n$7\r
\nMariaDB\r\n
*3\r\n$3\r\nSET\r\n$4\r\ndate\r\n$8\r\n2013-9-5\r\n
*3\r\n$4\r\nINCR\r\n$13\r\nclick_counter\r\n$5\r\n10086\r\n
```

如果这时flushAppendOnlyFile函数被调用，假设服务器当前appendfsync选项的值为everysec，并且距离上次同步AOF文件已经超过一秒钟，那么服务器会先将aof_buf中的内容写入到AOF文件中，然后再对AOF文件进行同步。

以上就是对AOF持久化功能的文件写入和文件同步这两个步骤的介绍。

AOF持久化的效率和安全性

服务器配置appendfsync选项的值直接决定AOF持久化功能的效率和安全性。

·当appendfsync的值为always时，服务器在每个事件循环都要将aof_buf缓冲区中的所有内容写入到AOF文件，并且同步AOF文件，所以always的效率是appendfsync选项三个值当中最慢的一个，但从安全性来说，always也是最安全的，因为即使出现故障停机，AOF持久化也只会丢失一个事件循环中所产生的命令数据。

·当appendfsync的值为everysec时，服务器在每个事件循环都要将aof_buf缓冲区中的所有内容写入到AOF文件，并且每隔一秒就要在子线程中对AOF文件进行一次同步。从效率上来讲，everysec模式足够快，并且就算出现故障停机，数据库也只丢失一秒钟的命

令数据。

·当appendfsync的值为no时，服务器在每个事件循环都要将aof_buf缓冲区中的所有内容写入到AOF文件，至于何时对AOF文件进行同步，则由操作系统控制。因为处于no模式下的flushAppendOnlyFile调用无须执行同步操作，所以该模式下的AOF文件写入速度总是最快的，不过因为这种模式会在系统缓存中积累一段时间的写入数据，所以该模式的单次同步时长通常是三种模式中时间最长的。从平摊操作的角度来看，no模式和everysec模式的效率类似，当出现故障停机时，使用no模式的服务器将丢失上次同步AOF文件之后的所有写命令数据。

11.2　AOF文件的载入与数据还原

因为AOF文件里面包含了重建数据库状态所需的所有写命令，所以服务器只要读入并重新执行一遍AOF文件里面保存的写命令，就可以还原服务器关闭之前的数据库状态。

Redis读取AOF文件并还原数据库状态的详细步骤如下：

1）创建一个不带网络连接的伪客户端（fake client）：因为Redis的命令只能在客户端上下文中执行，而载入AOF文件时所使用的命令直接来源于AOF文件而不是网络连接，所以服务器使用了一个没有网络连接的伪客户端来执行AOF文件保存的写命令，伪客户端执行命令的效果和带网络连接的客户端执行命令的效果完全一样。

2）从AOF文件中分析并读取出一条写命令。

3）使用伪客户端执行被读出的写命令。

4）一直执行步骤2和步骤3，直到AOF文件中的所有写命令都被处理完毕为止。

当完成以上步骤之后，AOF文件所保存的数据库状态就会被完整地还原出来，整个过程如图11-2所示。

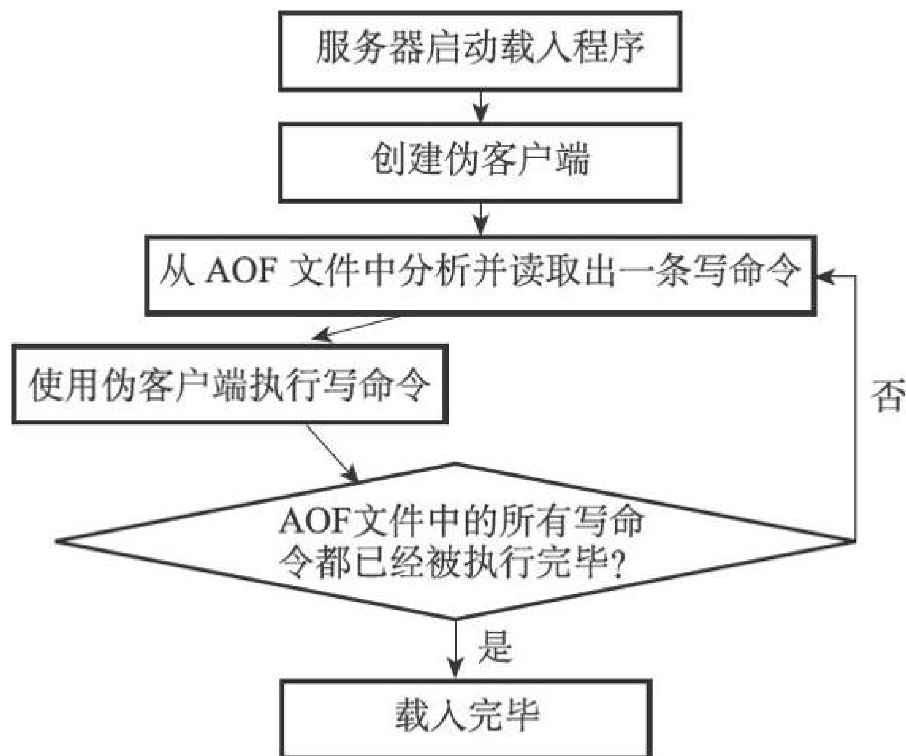


图11-2　AOF文件载入过程

例如，对于以下AOF文件来说：

```
*2\r\n$6\r\nSELECT\r\n$1\r\n0\r\n
*3\r\n$3\r\nSET\r\n$3\r\nmsg\r\n$5\r\nhello\r\n
*5\r\n$4\r\nSADD\r\n$6\r\nfruits\r\n$5\r\napple\r\n$6\r\nbanana\r\n$6\r
\ncherry\r\n
*5\r\n$5\r\nRPUSH\r\n$7\r\nnumbers\r\n$3\r\n128\r\n$3\r\n256\r\n$3\r\n512\r\n
```

服务器首先读入并执行SELECT 0命令，之后是SET msg hello命令，再之后是SADD fruits apple banana cherry命令，最后是RPUSH numbers 128 256 512命令，当这些命令都执行完毕之后，服务器的数据库就被还原到之前的状态了。

以上就是服务器读入AOF文件，并根据文件内容来还原数据库状态的原理。

11.3 AOF重写

因为AOF持久化是通过保存被执行的写命令来记录数据库状态的，所以随着服务器运行时间的流逝，AOF文件中的内容会越来越多，文件的体积也会越来越大，如果不加以控制的话，体积过大的AOF文件很可能对Redis服务器、甚至整个宿主计算机造成影响，并且AOF文件的体积越大，使用AOF文件来进行数据还原所需的时间就越多。

举个例子，如果客户端执行了以下命令：

```
redis> RPUSH list "A" "B"          // ["A", "B"]
(integer) 2
redis> RPUSH list "C"                        // ["A", "B", "C"]
(integer) 3
redis> RPUSH list "D" "E"            // ["A", "B", "C", "D", "E"]
(integer) 5
redis> LPOP list                     // ["B", "C", "D", "E"]
"A"
redis> LPOP list                     // ["C", "D", "E"]
"B"
redis> RPUSH list "F" "G"            // ["C", "D", "E", "F", "G"]
(integer) 5
```

那么光是为了记录这个list键的状态，AOF文件就需要保存六条命令。

对于实际的应用程度来说，写命令执行的次数和频率会比上面的简单示例要高得多，所以造成的问题也会严重得多。

为了解决AOF文件体积膨胀的问题，Redis提供了AOF文件重写（rewrite）功能。通过该功能，Redis服务器可以创建一个新的AOF文件来替代现有的AOF文件，新旧两个AOF文件所保存的数据库状态相同，但新AOF文件不会包含任何浪费空间的冗余命令，所以新AOF文件的体积通常会比旧AOF文件的体积要小得多。

在接下来的内容中，我们将介绍AOF文件重写的实现原理，以及BGREWEITEAOF命令的实现原理。

11.3.1 AOF文件重写的实现

虽然Redis将生成新AOF文件替换旧AOF文件的功能命名为“AOF文件重写”，但实际上，AOF文件重写并不需要对现有的AOF文件进行任何读取、分析或者写入操作，这个功能是通过读取服务器当前的数据库状态来实现的。

考虑这样一个情况，如果服务器对list键执行了以下命令：

```
redis> RPUSH list "A" "B"                 // ["A", "B"]
(integer) 2
```

```
redis> RPUSH list "C"                        // ["A", "B", "C"]
(integer) 3
redis> RPUSH list "D" "E"            // ["A", "B", "C", "D", "E"]
(integer) 5
redis> LPOP list                     // ["B", "C", "D", "E"]
"A"
redis> LPOP list                     // ["C", "D", "E"]
"B"
redis> RPUSH list "F" "G"            // ["C", "D", "E", "F", "G"]
(integer) 5
```

那么服务器为了保存当前list键的状态，必须在AOF文件中写入六条命令。

如果服务器想要用尽量少的命令来记录list键的状态，那么最简单高效的办法不是去读取和分析现有AOF文件的内容，而是直接从数据库中读取键list的值，然后用一条RPUSH list"C""D""E""F""G"命令来代替保存在AOF文件中的六条命令，这样就可以将保存list键所需的命令从六条减少为一条了。

再考虑这样一个例子，如果服务器对animals键执行了以下命令：

```
redis> SADD animals "Cat"
   // {"Cat"}
(integer) 1
redis> SADD animals "Dog" "Panda" "Tiger"    // {"Cat", "Dog", "Panda", "Tiger"}
(integer) 3
redis> SREM animals "Cat"                    // {"Dog", "Panda", "Tiger"}
(integer) 1
redis> SADD animals "Lion" "Cat"             // {"Dog", "Panda", "Tiger",
(integer) 2                                     "Lion", "Cat"}
```

那么为了记录animals键的状态，AOF文件必须保存上面列出的四条命令。

如果服务器想减少保存animals键所需命令的数量，那么服务器可以通过读取animals键的值，然后用一条SADD animals"Dog""Panda""Tiger""Lion""Cat"命令来代替上面的四条命令，这样就将保存animals键所需的命令从四条减少为一条了。

除了上面列举的列表键和集合键之外，其他所有类型的键都可以用同样的方法去减少AOF文件中的命令数量。首先从数据库中读取键现在的值，然后用一条命令去记录键值对，代替之前记录这个键值对的多条命令，这就是AOF重写功能的实现原理。

整个重写过程可以用以下伪代码表示：

```
def aof_rewrite(new_aof_file_name):
    #
创建新 AOF
文件
    f = create_file(new_aof_file_name)
    #
遍历数据库
    for db in redisServer.db:
```

```
        #
忽略空数据库
        if db.is_empty(): continue
        #
写入SELECT
命令，指定数据库号码
        f.write_command("SELECT" + db.id)
        #
遍历数据库中的所有键
        for key in db:
            #
忽略已过期的键
            if key.is_expired(): continue
            #
根据键的类型对键进行重写
            if key.type == String:
                rewrite_string(key)
            elif key.type == List:
                rewrite_list(key)
            elif key.type == Hash:
                rewrite_hash(key)
            elif key.type == Set:
                rewrite_set(key)
            elif key.type == SortedSet:
                rewrite_sorted_set(key)
            #
如果键带有过期时间，那么过期时间也要被重写
            if key.have_expire_time():
                rewrite_expire_time(key)
    #
写入完毕，关闭文件
    f.close()
def rewrite_string(key):
    #
使用GET
命令获取字符串键的值
    value = GET(key)
    #
使用SET
命令重写字符串键
    f.write_command(SET, key, value)
def rewrite_list(key):
    #
使用LRANGE
命令获取列表键包含的所有元素
    item1, item2, ..., itemN = LRANGE(key, 0, -1)
    #
使用RPUSH
命令重写列表键
    f.write_command(RPUSH, key, item1, item2, ..., itemN)
def rewrite_hash(key):
    #
使用HGETALL
命令获取哈希键包含的所有键值对
    field1, value1, field2, value2, ..., fieldN, valueN = HGETALL(key)
    #
使用HMSET
命令重写哈希键
    f.write_command(HMSET, key, field1, value1, field2, value2, ..., fieldN, valueN)
def rewrite_set(key);
    #
使用SMEMBERS
命令获取集合键包含的所有元素
    elem1, elem2, ..., elemN = SMEMBERS(key)
    #
使用SADD
命令重写集合键
    f.write_command(SADD, key, elem1, elem2, ..., elemN)
def rewrite_sorted_set(key):
    #
使用ZRANGE
```

```
命令获取有序集合键包含的所有元素
    member1, score1, member2, score2, ..., memberN, scoreN = ZRANGE(key, 0, -1, "WITH
    #
使用ZADD
命令重写有序集合键
    f.write_command(ZADD, key, score1, member1, score2, member2, ..., scoreN, memberN
def rewrite_expire_time(key):
    #
获取毫秒精度的键过期时间戳
    timestamp = get_expire_time_in_unixstamp(key)
    #
使用PEXPIREAT
命令重写键的过期时间
    f.write_command(PEXPIREAT, key, timestamp)
```

因为aof_rewrite函数生成的新AOF文件只包含还原当前数据库状态所必须的命令，所以新AOF文件不会浪费任何硬盘空间。

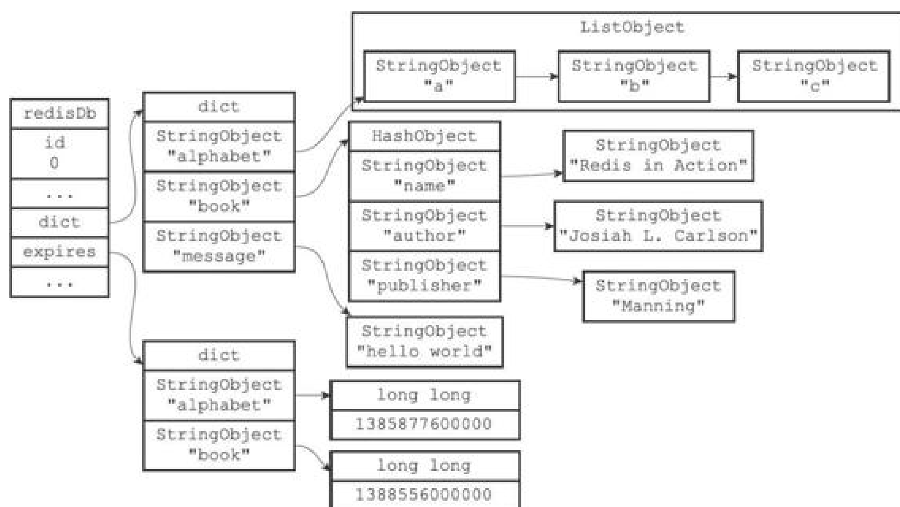
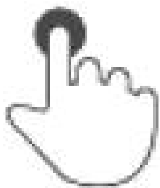


图11-3　一个数据库

例如，对于图11-3所示的数据库，aof_rewrite函数产生的新AOF文件将包含以下命令：

```
SELECT 0
RPUSH alphabet "a" "b" "c"
EXPIREAT alphabet 1385877600000
HMSET book "name" "Redisin Action"
           "author" "Josiah L. Carlson"
           "publisher" "Manning"
EXPIREAT book 1388556000000
SET message "hello world"
```

以上命令就是还原图11-3所示的数据库所必须的命令，它们没有一条是多余的。

注意

在实际中，为了避免在执行命令时造成客户端输入缓冲区溢出，重写程序在处理列表、哈希表、集合、有序集合这四种可能会带有多个元素的键时，会先检查键所包含的元素数量，如果元素的数量超过了redis.h/REDIS_AOF_REWRITE_ITEMS_PER_CMD常量的值，那么重写程序将使用多条命令来记录键的值，而不单单使用一条命令。

在目前版本中，REDIS_AOF_REWRITE_ITEMS_PER_CMD常量的值为64，这也就是说，如果一个集合键包含了超过64个元素，那么重写程序会用多条SADD命令来记录这个集合，并且每条命令设置的元素数量也为64个：

```
SADD <set-key> <elem1> <elem2> ... <elem64>
SADD <set-key> <elem65> <elem66> ... <elem128>
SADD <set-key> <elem129> <elem130> ... <elem192>
...
```

另一方面如果一个列表键包含了超过64个项，那么重写程序会用多条RPUSH命令来保存这个列表，并且每条命令设置的项数量也为64个：

```
RPUSH <list-key> <item1> <item2> ... <item64>
RPUSH <list-key> <item65> <item66> ... <item128>
RPUSH <list-key> <item129> <item130> ... <item192>
...
```

重写程序使用类似的方法处理包含多个元素的有序集合键，以及包含多个键值对的哈希表键。

11.3.2　AOF后台重写

上面介绍的AOF重写程序aof_rewrite函数可以很好地完成创建一个新AOF文件的任务，但是，因为这个函数会进行大量的写入操作，所以调用这个函数的线程将被长时间阻塞，因为Redis服务器使用单个线程来处理命令请求，所以如果由服务器直接调用aof_rewrite函数的话，那么在重写AOF文件期间，服务期将无法处理客户端发来的命令请求。

很明显，作为一种辅佐性的维护手段，Redis不希望AOF重写造成服务器无法处理请求，所以Redis决定将AOF重写程序放到子进程里执行，这样做可以同时达到两个目的：

·子进程进行AOF重写期间，服务器进程（父进程）可以继续处理命令请求。

·子进程带有服务器进程的数据副本，使用子进程而不是线程，可以在避免使用锁的情况下，保证数据的安全性。

不过，使用子进程也有一个问题需要解决，因为子进程在进行AOF重写期间，服务器进程还需要继续处理命令请求，而新的命令可能会对现有的数据库状态进行修改，从而使得服务器当前的数据库状态和重写后的AOF文件所保存的数据库状态不一致。

表11-2展示了一个AOF文件重写例子，当子进程开始进行文件重写时，数据库中只有k1一个键，但是当子进程完成AOF文件重写之后，服务器进程的数据库中已经新设置了k2、k3、k4三个键，因此，重写后的AOF文件和服务器当前的数据库状态并不一致，新的AOF文件只保存了k1一个键的数据，而服务器数据库现在却有k1、k2、k3、k4四个键。

表11-2　AOF文件重写时的服务器进程和子进程

时间	服务器进程	子进程
T1	执行命令 SET k1 v1	
T2	执行命令 SET k1 v2	
T3	执行命令 SET k1 v3	
T4	创建子进程，执行 AOF 文件重写	开始 AOF 文件重写
T5	执行命令 SET k2 10086	执行重写操作
T6	执行命令 SET k3 12345	执行重写操作
T7	执行命令 SET k4 22222	完成 AOF 文件重写

为了解决这种数据不一致问题，Redis服务器设置了一个AOF重写缓冲区，这个缓冲区在服务器创建子进程之后开始使用，当Redis服务器执行完一个写命令之后，它会同时将这个写命令发送给AOF缓冲区和AOF重写缓冲区，如图11-4所示。

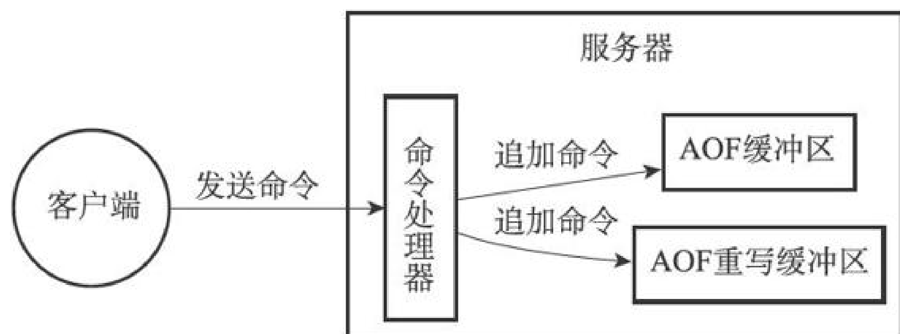


图11-4　服务器同时将命令发送给AOF文件和AOF重写缓冲区

这也就是说，在子进程执行AOF重写期间，服务器进程需要执行以下三个工作：

1）执行客户端发来的命令。

2）将执行后的写命令追加到AOF缓冲区。

3）将执行后的写命令追加到AOF重写缓冲区。

这样一来可以保证：

·AOF缓冲区的内容会定期被写入和同步到AOF文件，对现有AOF文件的处理工作会如常进行。

·从创建子进程开始，服务器执行的所有写命令都会被记录到AOF重写缓冲区里面。

当子进程完成AOF重写工作之后，它会向父进程发送一个信号，父进程在接到该信号之后，会调用一个信号处理函数，并执行以下工作：

1）将AOF重写缓冲区中的所有内容写入到新AOF文件中，这时新AOF文件所保存的数据库状态将和服务器当前的数据库状态一致。

2）对新的AOF文件进行改名，原子地（atomic）覆盖现有的AOF文件，完成新旧两个AOF文件的替换。

这个信号处理函数执行完毕之后，父进程就可以继续像往常一样接受命令请求了。

在整个AOF后台重写过程中，只有信号处理函数执行时会对服务器进程（父进程）造成阻塞，在其他时候，AOF后台重写都不会阻塞父进程，这将AOF重写对服务器性能造成的影响降到了最低。

举个例子，表11-3展示了一个AOF文件后台重写的执行过程：

·当子进程开始重写时，服务器进程（父进程）的数据库中只有k1一个键，当子进程完成AOF文件重写之后，服务器进程的数据库中已经多出了k2、k3、k4三个新键。

·在子进程向服务器进程发送信号之后，服务器进程会将保存在AOF重写缓冲区里面记录的k2、k3、k4三个键的命令追加到新AOF文件的末尾，然后用新AOF文件替换旧AOF文件，完成AOF文件后台重写操作。

表11-3　AOF文件后台重写过程

时间	服务器进程（父进程）	子进程
T1	执行命令 SET k1 v1	
T2	执行命令 SET k1 v2	
T3	执行命令 SET k1 v3	
T4	创建子进程，执行 AOF 文件重写	开始 AOF 文件重写
T5	执行命令 SET k2 10086	执行重写操作
T6	执行命令 SET k3 12345	执行重写操作
T7	执行命令 SET k4 22222	完成 AOF 文件重写，向父进程发送信号
T8	接收到子进程发来的信号，将命令 SET k2 10086、SET k3 12345、SET k4 22222 追加到新 AOF 文件的末尾	
T9	用新 AOF 文件覆盖旧 AOF 文件	

以上就是AOF后台重写，也即是BGREWRITEAOF命令的实现原理。

11.4 重点回顾

·AOF文件通过保存所有修改数据库的写命令请求来记录服务器的数据库状态。

·AOF文件中的所有命令都以Redis命令请求协议的格式保存。

·命令请求会先保存到AOF缓冲区里面，之后再定期写入并同步到AOF文件。

·appendfsync选项的不同值对AOF持久化功能的安全性以及Redis服务器的性能有很大的影响。

·服务器只要载入并重新执行保存在AOF文件中的命令，就可以还原数据库本来的状态。

·AOF重写可以产生一个新的AOF文件，这个新的AOF文件和原有的AOF文件所保存的数据库状态一样，但体积更小。

·AOF重写是一个有歧义的名字，该功能是通过读取数据库中的键值对来实现的，程序无须对现有AOF文件进行任何读入、分析或者写入操作。

·在执行BGREWRITEAOF命令时，Redis服务器会维护一个AOF重写缓冲区，该缓冲区会在子进程创建新AOF文件期间，记录服务器执行的所有写命令。当子进程完成创建新AOF文件的工作之后，服务器会将重写缓冲区中的所有内容追加到新AOF文件的末尾，使得新旧两个AOF文件所保存的数据库状态一致。最后，服务器用新的AOF文件替换旧的AOF文件，以此来完成AOF文件重写操作。

第12章　事件

Redis服务器是一个事件驱动程序，服务器需要处理以下两类事件：

·文件事件（file event）：Redis服务器通过套接字与客户端（或者其他Redis服务器）进行连接，而文件事件就是服务器对套接字操作的抽象。服务器与客户端（或者其他服务器）的通信会产生相应的文件事件，而服务器则通过监听并处理这些事件来完成一系列网络通信操作。

·时间事件（time event）：Redis服务器中的一些操作（比如serverCron函数）需要在给定的时间点执行，而时间事件就是服务器对这类定时操作的抽象。

本章将对文件事件和时间事件进行介绍，说明这两种事件在Redis服务器中的应用，它们的实现方法，以及处理这些事件的API等等。

本章最后将对服务器的事件调度方式进行介绍，说明Redis服务器是如何安排并执行文件事件和时间事件的。

12.1 文件事件

Redis基于Reactor模式开发了自己的网络事件处理器：这个处理器被称为文件事件处理器（file event handler）：

·文件事件处理器使用I/O多路复用（multiplexing）程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。

·当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时，与操作相对应的文件事件就会产生，这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

虽然文件事件处理器以单线程方式运行，但通过使用I/O多路复用程序来监听多个套接字，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与Redis服务器中其他同样以单线程方式运行的模块进行对接，这保持了Redis内部单线程设计的简单性。

12.1.1 文件事件处理器的构成

图12-1展示了文件事件处理器的四个组成部分，它们分别是套接字、I/O多路复用程序、文件事件分派器（dispatcher），以及事件处理器。

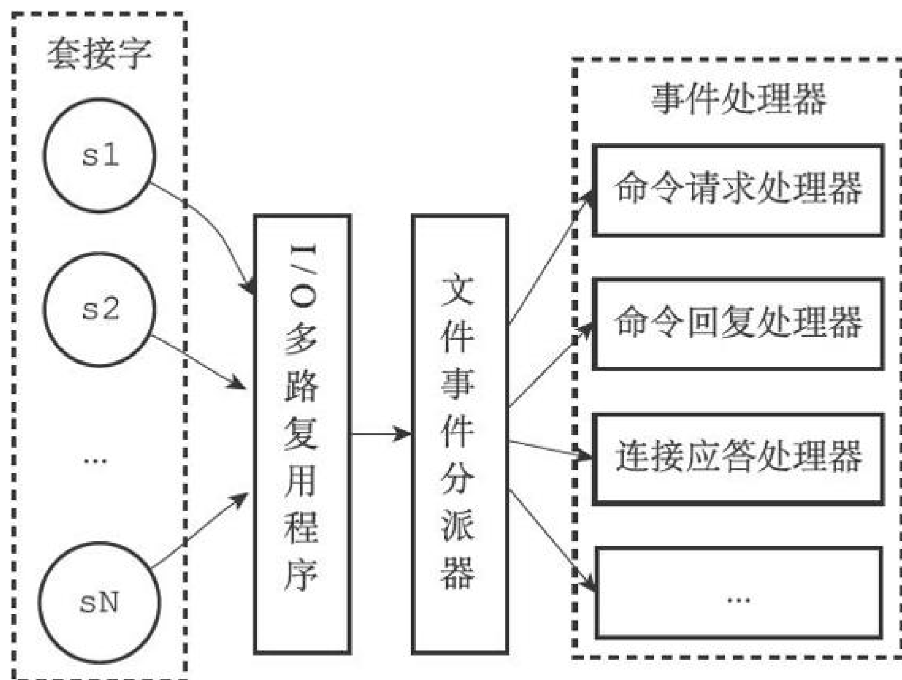


图12-1　文件事件处理器的四个组成部分

文件事件是对套接字操作的抽象，每当一个套接字准备好执行连接应答（accept）、写入、读取、关闭等操作时，就会产生一个文件事件。因为一个服务器通常会连接多个套接字，所以多个文件事件有可能会并发地出现。

I/O多路复用程序负责监听多个套接字，并向文件事件分派器传送那些产生了事件的套接字。

尽管多个文件事件可能会并发地出现，但I/O多路复用程序总是会将所有产生事件的套接字都放到一个队列里面，然后通过这个队列，以有序（sequentially）、同步（synchronously）、每次一个套接字的方式向文件事件分派器传送套接字。当上一个套接字产生的事件被处理完毕之后（该套接字为事件所关联的事件处理器执行完毕），I/O多路复用程序才会继续向文件事件分派器传送下一个套接字，如图12-2所示。

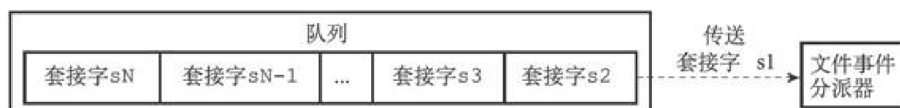


图12-2　I/O多路复用程序通过队列向文件事件分派器传送套接字

文件事件分派器接收I/O多路复用程序传来的套接字，并根据套接字产生的事件的类型，调用相应的事件处理器。

服务器会为执行不同任务的套接字关联不同的事件处理器，这些处理器是一个个函数，它们定义了某个事件发生时，服务器应该执行的动作。

12.1.2　I/O多路复用程序的实现

Redis的I/O多路复用程序的所有功能都是通过包装常见的select、epoll、evport和kqueue这些I/O多路复用函数库来实现的，每个I/O多路复用函数库在Redis源码中都对应一个单独的文件，比如ae_select.c、ae_epoll.c、ae_kqueue.c，诸如此类。

因为Redis为每个I/O多路复用函数库都实现了相同的API，所以I/O多路复用程序的底层实现是可以互换的，如图12-3所示。

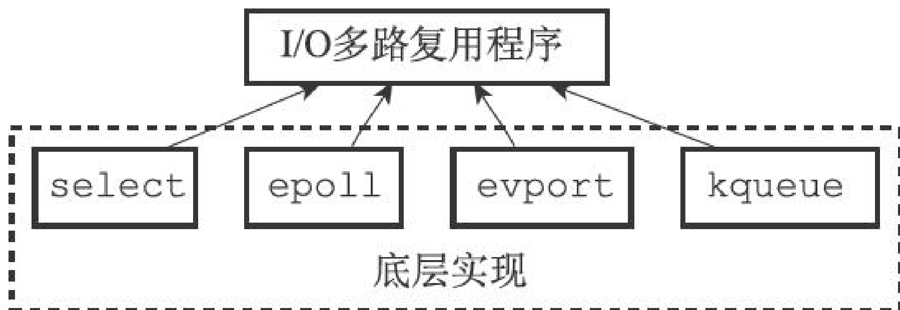


图12-3　Redis的I/O多路复用程序有多个I/O多路复用库实现可选

Redis在I/O多路复用程序的实现源码中用#include宏定义了相应的规则，程序会在编译时自动选择系统中性能最高的I/O多路复用函数库来作为Redis的I/O多路复用程序的底层实现：

```
/* Include the best multiplexing layer supported by this system.
  * The following should be ordered by performances, descending. */
# ifdef HAVE_EVPORT
# include "ae_evport.c"
# else
    # ifdef HAVE_EPOLL
    # include "ae_epoll.c"
    # else
        # ifdef HAVE_KQUEUE
        # include "ae_kqueue.c"
        # else
        # include "ae_select.c"
        # endif
    # endif
# endif
```

12.1.3　事件的类型

I/O多路复用程序可以监听多个套接字的ae.h/AE_READABLE事件和ae.h/AE_WRITABLE事件，这两类事件和套接字操作之间的对应关系如下：

·当套接字变得可读时（客户端对套接字执行write操作，或者执行close操作），或者有新的可应答（acceptable）套接字出现时（客户端对服务器的监听套接字执行connect操作），套接字产生AE_READABLE事件。

·当套接字变得可写时（客户端对套接字执行read操作），套接字产生AE_WRITABLE事件。

I/O多路复用程序允许服务器同时监听套接字的AE_READABLE事件和AE_WRITABLE事件，如果一个套接字同时产生了这两种事件，那么文件事件分派器会优先处理AE_READABLE事件，等到AE_READABLE事件处理完之后，才处理AE_WRITABLE事件。

这也就是说，如果一个套接字又可读又可写的话，那么服务器将先读套接字，后写套接字。

12.1.4 API

ae.c/aeCreateFileEvent函数接受一个套接字描述符、一个事件类型，以及一个事件处理器作为参数，将给定套接字的给定事件加入到I/O多路复用程序的监听范围之内，并对事件和事件处理器进行关联。

ae.c/aeDeleteFileEvent函数接受一个套接字描述符和一个监听事件类型作为参数，让I/O多路复用程序取消对给定套接字的给定事件的监听，并取消事件和事件处理器之间的关联。

ae.c/aeGetFileEvents函数接受一个套接字描述符，返回该套接字正在被监听的事件类型：

·如果套接字没有任何事件被监听，那么函数返回AE_NONE。

·如果套接字的读事件正在被监听，那么函数返回AE_READABLE。

·如果套接字的写事件正在被监听，那么函数返回AE_WRITABLE。

·如果套接字的读事件和写事件正在被监听，那么函数返回AE_READABLE|AE_WRITABLE。

ae.c/aeWait函数接受一个套接字描述符、一个事件类型和一个毫秒数为参数，在给定的时间内阻塞并等待套接字的给定类型事件产生，当事件成功产生，或者等待超时之后，函数返回。

ae.c/aeApiPoll函数接受一个sys/time.h/struct timeval结构为参数，并在指定的时间内，阻塞并等待所有被aeCreateFileEvent函数设置为监听状态的套接字产生文件事件，当有至少一个事件产生，或者等待超时后，函数返回。

ae.c/aeProcessEvents函数是文件事件分派器，它先调用aeApiPoll函数来等待事件产生，然后遍历所有已产生的事件，并调用相应的事件处理器来处理这些事件。

ae.c/aeGetApiName函数返回I/O多路复用程序底层所使用的I/O多路复用函数库的名称：返回"epoll"表示底层为epoll函数库，返回"select"表示底层为select函数库，诸如此类。

12.1.5 文件事件的处理器

Redis为文件事件编写了多个处理器，这些事件处理器分别用于实现不同的网络通信需求，比如说：

·为了对连接服务器的各个客户端进行应答，服务器要为监听套接字关联连接应答处理器。

·为了接收客户端传来的命令请求，服务器要为客户端套接字关联命令请求处理器。

·为了向客户端返回命令的执行结果，服务器要为客户端套接字关联命令回复处理器。

·当主服务器和从服务器进行复制操作时，主从服务器都需要关联特别为复制功能编写的复制处理器。

在这些事件处理器里面，服务器最常用的要数与客户端进行通信的连接应答处理器、命令请求处理器和命令回复处理器。

1.连接应答处理器

networking.c/acceptTcpHandler函数是Redis的连接应答处理器，这个处理器用于对连接服务器监听套接字的客户端进行应答，具体实现为sys/socket.h/accept函数的包装。

当Redis服务器进行初始化的时候，程序会将这个连接应答处理器和服务器监听套接字的AE_READABLE事件关联起来，当有客户端用sys/socket.h/connect函数连接服务器监听套接字的时候，套接字就会产生AE_READABLE事件，引发连接应答处理器执行，并执行相应的套接字应答操作，如图12-4所示。

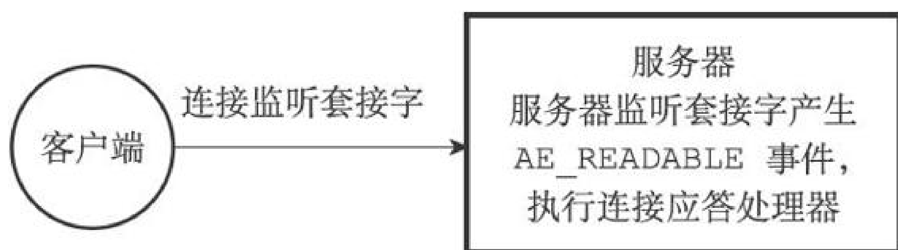


图12-4　服务器对客户端的连接请求进行应答

2.命令请求处理器

networking.c/readQueryFromClient函数是Redis的命令请求处理器，这个处理器负责从套接字中读入客户端发送的命令请求内容，具体实现为unistd.h/read函数的包装。

当一个客户端通过连接应答处理器成功连接到服务器之后，服务器会将客户端套接字的AE_READABLE事件和命令请求处理器关联起来，当客户端向服务器发送命令请求的时候，套接字就会产生AE_READABLE事件，引发命令请求处理器执行，并执行相应的套接字

读入操作，如图12-5所示。

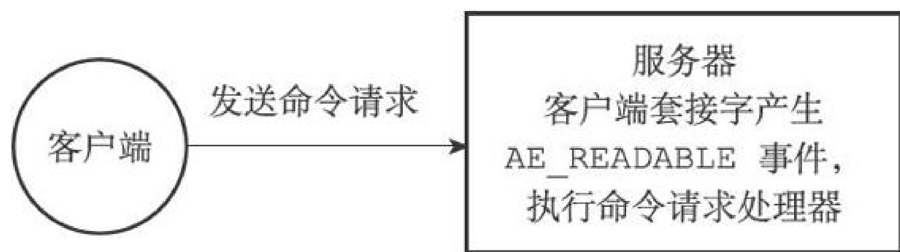


图12-5　服务器接收客户端发来的命令请求

在客户端连接服务器的整个过程中，服务器都会一直为客户端套接字的AE_READABLE事件关联命令请求处理器。

3.命令回复处理器

networking.c/sendReplyToClient函数是Redis的命令回复处理器，这个处理器负责将服务器执行命令后得到的命令回复通过套接字返回给客户端，具体实现为unistd.h/write函数的包装。

当服务器有命令回复需要传送给客户端的时候，服务器会将客户端套接字的AE_WRITABLE事件和命令回复处理器关联起来，当客户端准备好接收服务器传回的命令回复时，就会产生AE_WRITABLE事件，引发命令回复处理器执行，并执行相应的套接字写入操作，如图12-6所示。



图12-6　服务器向客户端发送命令回复

当命令回复发送完毕之后，服务器就会解除命令回复处理器与客户端套接字的AE_WRITABLE事件之间的关联。

4.一次完整的客户端与服务器连接事件示例

让我们来追踪一次Redis客户端与服务器进行连接并发送命令的整个过程，看看在过程中会产生什么事件，而这些事件又是如何被处理的。

假设一个Redis服务器正在运作，那么这个服务器的监听套接字的AE_READABLE事件应该正处于监听状态之下，而该事件所对应的处理

器为连接应答处理器。

如果这时有一个Redis客户端向服务器发起连接，那么监听套接字将产生AE_READABLE事件，触发连接应答处理器执行。处理器会对客户端的连接请求进行应答，然后创建客户端套接字，以及客户端状态，并将客户端套接字的AE_READABLE事件与命令请求处理器进行关联，使得客户端可以向主服务器发送命令请求。

之后，假设客户端向主服务器发送一个命令请求，那么客户端套接字将产生AE_READABLE事件，引发命令请求处理器执行，处理器读取客户端的命令内容，然后传给相关程序去执行。

执行命令将产生相应的命令回复，为了将这些命令回复传送回客户端，服务器会将客户端套接字的AE_WRITABLE事件与命令回复处理器进行关联。当客户端尝试读取命令回复的时候，客户端套接字将产生AE_WRITABLE事件，触发命令回复处理器执行，当命令回复处理器将命令回复全部写入到套接字之后，服务器就会解除客户端套接字的AE_WRITABLE事件与命令回复处理器之间的关联。

图12-7总结了上面描述的整个通信过程，以及通信时用到的事件处理器。

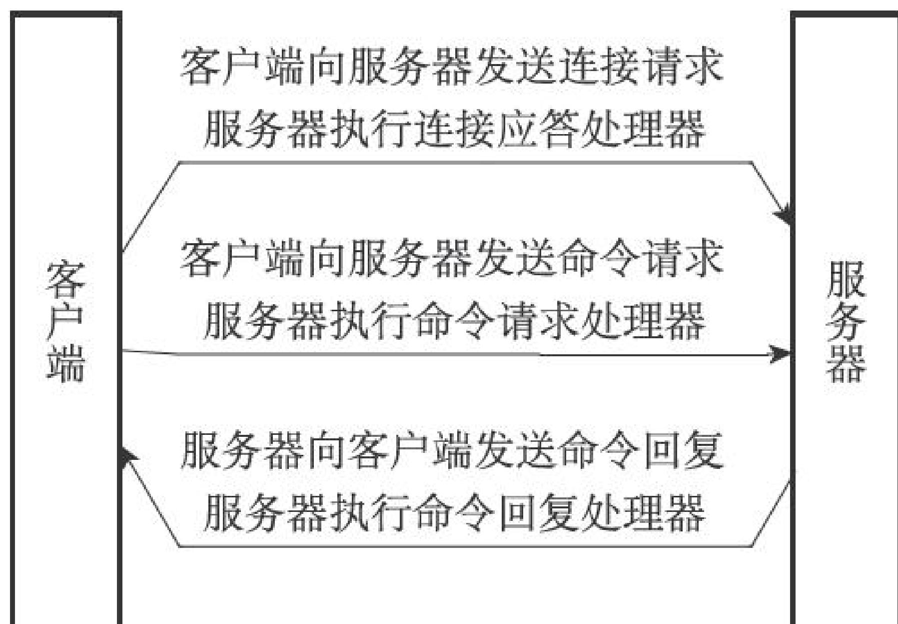


图12-7　客户端和服务器的通信过程

12.2 时间事件

Redis的时间事件分为以下两类：

·定时事件：让一段程序在指定的时间之后执行一次。比如说，让程序X在当前时间的30毫秒之后执行一次。

·周期性事件：让一段程序每隔指定时间就执行一次。比如说，让程序Y每隔30毫秒就执行一次。

一个时间事件主要由以下三个属性组成：

·id：服务器为时间事件创建的全局唯一ID（标识号）。ID号按从小到大的顺序递增，新事件的ID号比旧事件的ID号要大。

·when：毫秒精度的UNIX时间戳，记录了时间事件的到达（arrive）时间。

·timeProc：时间事件处理器，一个函数。当时间事件到达时，服务器就会调用相应的处理器来处理事件。

一个时间事件是定时事件还是周期性事件取决于时间事件处理器的返回值：

·如果事件处理器返回ae.h/AE_NOMORE，那么这个事件为定时事件：该事件在达到一次之后就会被删除，之后不再到达。

·如果事件处理器返回一个非AE_NOMORE的整数值，那么这个事件为周期性时间：当一个时间事件到达之后，服务器会根据事件处理器返回的值，对时间事件的when属性进行更新，让这个事件在一段时间之后再次到达，并以这种方式一直更新并运行下去。比如说，如果一个时间事件的处理器返回整数值30，那么服务器应该对这个时间事件进行更新，让这个事件在30毫秒之后再次到达。

目前版本的Redis只使用周期性事件，而没有使用定时事件。

12.2.1 实现

服务器将所有时间事件都放在一个无序链表中，每当时间事件执行器运行时，它就遍历整个链表，查找所有已到达的时间事件，并调用相应的事件处理器。

图12-8展示了一个保存时间事件的链表的例子，链表中包含了三个不同的时间事件：因为新的时间事件总是插入到链表的表头，所以三个时间事件分别按ID逆序排序，表头事件的ID为3，中间事件的ID为2，表

尾事件的ID为1。



图12-8 用链表连接起来的三个时间事件

注意，我们说保存时间事件的链表为无序链表，指的不是链表不按ID排序，而是说，该链表不按when属性的大小排序。正因为链表没有按when属性进行排序，所以当时间事件执行器运行的时候，它必须遍历链表中的所有时间事件，这样才能确保服务器中所有已到达的时间事件都会被处理。

无序链表并不影响时间事件处理器的性能

在目前版本中，正常模式下的Redis服务器只使用serverCron一个时间事件，而在benchmark模式下，服务器也只使用两个时间事件。在这种情况下，服务器几乎是将无序链表退化成一个指针来使用，所以使用无序链表来保存时间事件，并不影响事件执行的性能。

12.2.2 API

ae.c/aeCreateTimeEvent函数接受一个毫秒数milliseconds和一个时间事件处理器proc作为参数，将一个新的时间事件添加到服务器，这个新的时间事件将在当前时间的milliseconds毫秒之后到达，而事件的处理器为proc。

例如，如果服务器当前所保存的时间事件如图12-9所示。

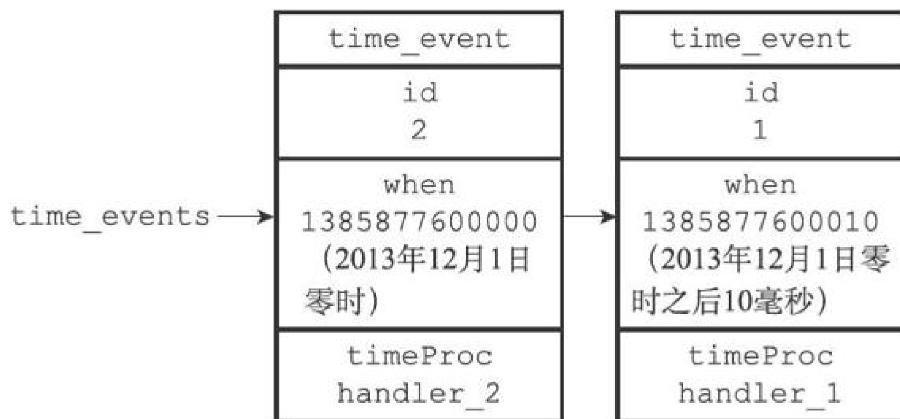


图12-9　用链表连接起来的两个时间事件

那么当程序以50毫秒和handler_3处理器为参数，在时间1385877599980（2013年12月1日零时前20毫秒）时调用aeCreateTimeEvent函数，服务器将创建ID为3的时间事件，这时服务器所保存的时间事件将如图12-8所示。

ae.c/aeDeleteFileEvent函数接受一个时间事件ID作为参数，然后从服务器中删除该ID所对应的时间事件。

举个例子，如果服务器当前保存的时间事件如图12-8所示，那么当程序调用aeDeleteFileEvent（3）之后，服务器保存的时间事件将变成图12-9所示的样子。

ae.c/aeSearchNearestTimer函数返回到达时间距离当前时间最接近的那个时间事件。

举个例子，如果当前时间为1385877599980（2013年12月1日零时前20毫秒），而服务器当前保存的时间事件如图12-8所示，那么调用aeSearchNearestTimer函数将返回ID为2的事件。

ae.c/processTimeEvents函数是时间事件的执行器，这个函数会遍历所有已到达的时间事件，并调用这些事件的处理器。已到达指的是，时间事件的when属性记录的UNIX时间戳等于或小于当前时间的UNIX时间戳。

举个例子，如果服务器保存的时间事件如图12-8所示，并且当前时间为1385877600010（2013年12月1日零时之后10毫秒），那么processTimeEvents函数将处理图中ID为2和1的时间事件，因为这两个事件的到达时间都大于等于1385877600010。

processTimeEvents函数的定义可以用以下伪代码来描述：

```
def processTimeEvents():
    #
遍历服务器中的所有时间事件
    for time_event in all_time_event():
        #
检查事件是否已经到达
        if time_event.when <= unix_ts_now():
            #
事件已到达
            #
执行事件处理器，并获取返回值
            retval = time_event.timeProc()
            #
如果这是一个定时事件
            if retval == AE_NOMORE:
                #
那么将该事件从服务器中删除
                delete_time_event_from_server(time_event)
        #
如果这是一个周期性事件
        else:
            #
那么按照事件处理器的返回值更新时间事件的 when
属性
            #
让这个事件在指定的时间之后再次到达
            update_when(time_event, retval)
```

12.2.3　时间事件应用实例：serverCron函数

持续运行的Redis服务器需要定期对自身的资源和状态进行检查和调整，从而确保服务器可以长期、稳定地运行，这些定期操作由redis.c/serverCron函数负责执行，它的主要工作包括：

·更新服务器的各类统计信息，比如时间、内存占用、数据库占用情况等。

·清理数据库中的过期键值对。

·关闭和清理连接失效的客户端。

·尝试进行AOF或RDB持久化操作。

·如果服务器是主服务器，那么对从服务器进行定期同步。

·如果处于集群模式，对集群进行定期同步和连接测试。

Redis服务器以周期性事件的方式来运行serverCron函数，在服务器运行期间，每隔一段时间，serverCron就会执行一次，直到服务器关闭为止。

在Redis2.6版本，服务器默认规定serverCron每秒运行10次，平均每间隔100毫秒运行一次。

从Redis2.8开始，用户可以通过修改hz选项来调整serverCron的每秒执行次数，具体信息请参考示例配置文件redis.conf关于hz选项的说

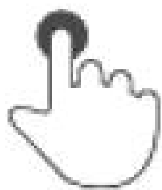
明。

12.3　事件的调度与执行

因为服务器中同时存在文件事件和时间事件两种事件类型，所以服务器必须对这两种事件进行调度，决定何时应该处理文件事件，何时又应该处理时间事件，以及花多少时间来处理它们等等。

事件的调度和执行由ae.c/aeProcessEvents函数负责，以下是该函数的伪代码表示：

```
def aeProcessEvents():
    #
获取到达时间离当前时间最接近的时间事件
    time_event = aeSearchNearestTimer()
    #
计算最接近的时间事件距离到达还有多少毫秒
    remaind_ms = time_event.when - unix_ts_now()
    #
如果事件已到达，那么remaind_ms
的值可能为负数，将它设定为0
    if remaind_ms < 0:
        remaind_ms = 0
    #
根据remaind_ms
的值，创建timeval
结构
    timeval = create_timeval_with_ms(remaind_ms)
    #
阻塞并等待文件事件产生，最大阻塞时间由传入的timeval
结构决定
    #
如果remaind_ms
的值为0
，那么aeApiPoll
调用之后马上返回，不阻塞
    aeApiPoll(timeval)
    #
处理所有已产生的文件事件
    processFileEvents()
    #
处理所有已到达的时间事件
    processTimeEvents()
```


注意

前面的12.1节在介绍文件事件API的时候，并没有讲到processFileEvents这个函数，因为它并不存在，在实际中，处理已产生文件事件的代码是直接写在aeProcessEvents函数里面的，这里为了方便讲述，才虚构了processFileEvents函数。

将aeProcessEvents函数置于一个循环里面，加上初始化和清理函数，这就构成了Redis服务器的主函数，以下是该函数的伪代码表示：

```
def main():
    #
初始化服务器
    init_server()
    #
一直处理事件，直到服务器关闭为止
    while server_is_not_shutdown():
        aeProcessEvents()
    #
服务器关闭，执行清理操作
    clean_server()
```

从事件处理的角度来看，Redis服务器的运行流程可以用流程图12-10来概括。

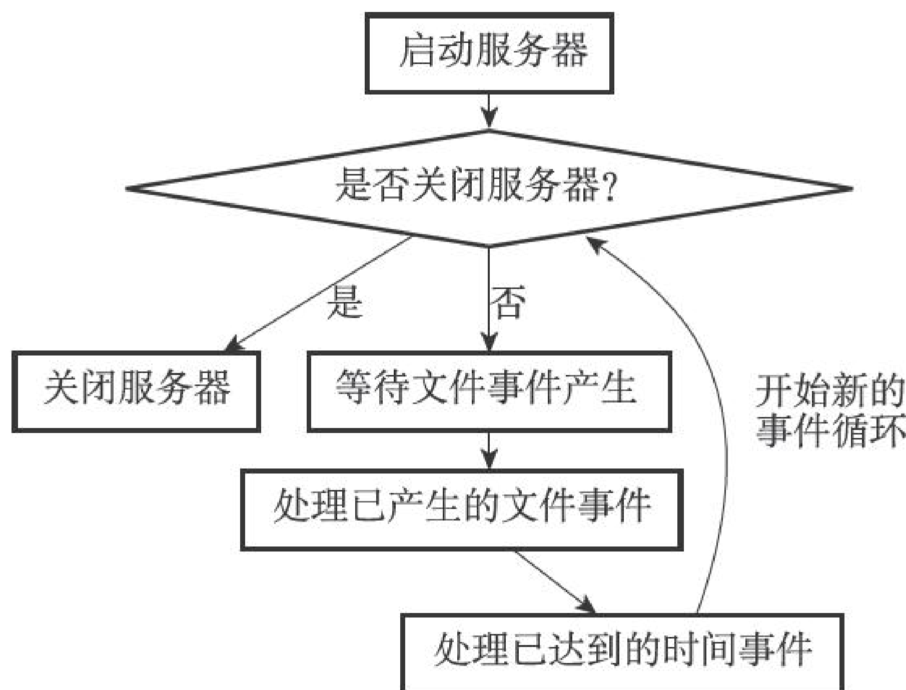


图12-10　事件处理角度下的服务器运行流程

以下是事件的调度和执行规则：

1）aeApiPoll函数的最大阻塞时间由到达时间最接近当前时间的时间事件决定，这个方法既可以避免服务器对时间事件进行频繁的轮询（忙等待），也可以确保aeApiPoll函数不会阻塞过长时间。

2）因为文件事件是随机出现的，如果等待并处理完一次文件事件之后，仍未有任何时间事件到达，那么服务器将再次等待并处理文件事件。随着文件事件的不断执行，时间会逐渐向时间事件所设置的到达时间逼近，并最终来到到达时间，这时服务器就可以开始处理到达的时间事件了。

3）对文件事件和时间事件的处理都是同步、有序、原子地执行的，服务器不会中途中断事件处理，也不会对事件进行抢占，因此，不管是文件事件的处理器，还是时间事件的处理器，它们都会尽可地减少程序的阻塞时间，并在有需要时主动让出执行权，从而降低造成事件饥饿的可能性。比如说，在命令回复处理器将一个命令回复写入到客户端套接字时，如果写入字节数超过了一个预设常量的话，命令回复处理器就会主动用break跳出写入循环，将余下的数据留到下次再写；另外，时间事件也会将非常耗时的持久化操作放到子线程或者子进程执行。

4）因为时间事件在文件事件之后执行，并且事件之间不会出现抢占，所以时间事件的实际处理时间，通常会比时间事件设定的到达时间稍晚一些。

表12-1记录了一次完整的事件调度和执行过程。

表12-1 一次完整的事件调度和执行过程

开始时间	结束时间	动作
0	10	创建一个在 100 毫秒到达的时间事件
11	30	等待文件事件
31	50	处理文件事件
51	85	等待文件事件
85	130	处理文件事件
131	150	执行时间事件

表12-1记录的事件执行过程凸显了上面列举的事件调度规则中的规则2、3、4：

·因为时间事件尚未到达，所以在处理时间事件之前，服务器已经等待并处理了两次文件事件。

·因为处理事件的过程中不会出现抢占，所以实际处理时间事件的时间比预定的100毫秒慢了30毫秒。

12.4　重点回顾

·Redis服务器是一个事件驱动程序，服务器处理的事件分为时间事件和文件事件两类。

·文件事件处理器是基于Reactor模式实现的网络通信程序。

·文件事件是对套接字操作的抽象：每次套接字变为可应答（acceptable）、可写（writable）或者可读（readable）时，相应的文件事件就会产生。

·文件事件分为AE_READABLE事件（读事件）和AE_WRITABLE事件（写事件）两类。

·时间事件分为定时事件和周期性事件：定时事件只在指定的时间到达一次，而周期性事件则每隔一段时间到达一次。

·服务器在一般情况下只执行serverCron函数一个时间事件，并且这个事件是周期性事件。

·文件事件和时间事件之间是合作关系，服务器会轮流处理这两种事件，并且处理事件的过程中也不会进行抢占。

·时间事件的实际处理时间通常会比设定的到达时间晚一些。

12.5 参考资料

·《Pattern-Oriented Software Architecture，Volume 4:A Pattern Language for Distributed Computing》第11章中的《Reactor》一节介绍了Reactor模型的定义、实现方法和作用。

·《Linux System Programming，Second Edition》第2章的《Multiplexed I/O》小节和第4章的《Event Poll》小节，以及《Unix环境高级编程，第2版》的14.5节，都对I/O多路复用及其相关函数进行了介绍。

第13章　客户端

Redis服务器是典型的一对多服务器程序：一个服务器可以与多个客户端建立网络连接，每个客户端可以向服务器发送命令请求，而服务器则接收并处理客户端发送的命令请求，并向客户端返回命令回复。

通过使用由I/O多路复用技术实现的文件事件处理器，Redis服务器使用单线程单进程的方式来处理命令请求，并与多个客户端进行网络通信。

对于每个与服务器进行连接的客户端，服务器都为这些客户端建立了相应的redis.h/redisClient结构（客户端状态），这个结构保存了客户端当前的状态信息，以及执行相关功能时需要用到的数据结构，其中包括：

·客户端的套接字描述符。

·客户端的名字。

·客户端的标志值（flag）。

·指向客户端正在使用的数据库的指针，以及该数据库的号码。

·客户端当前要执行的命令、命令的参数、命令参数的个数，以及指向命令实现函数的指针。

·客户端的输入缓冲区和输出缓冲区。

·客户端的复制状态信息，以及进行复制所需的数据结构。

·客户端执行BRPOP、BLPOP等列表阻塞命令时使用的数据结构。

·客户端的事务状态，以及执行WATCH命令时用到的数据结构。

·客户端执行发布与订阅功能时用到的数据结构。

·客户端的身份验证标志。

·客户端的创建时间，客户端和服务器最后一次通信的时间，以及客户端的输出缓冲区大小超出软性限制（soft limit）的时间。

Redis服务器状态结构的clients属性是一个链表，这个链表保存了所有与服务器连接的客户端的状态结构，对客户端执行批量操作，或者查找某个指定的客户端，都可以通过遍历clients链表来完成：

```
struct redisServer {
```

```
    // ...
    //
一个链表，保存了所有客户端状态
    list *clients;
    // ...
};
```

作为例子，图13-1展示了一个与三个客户端进行连接的服务器，而图13-2则展示了这个服务器的clients链表的样子。

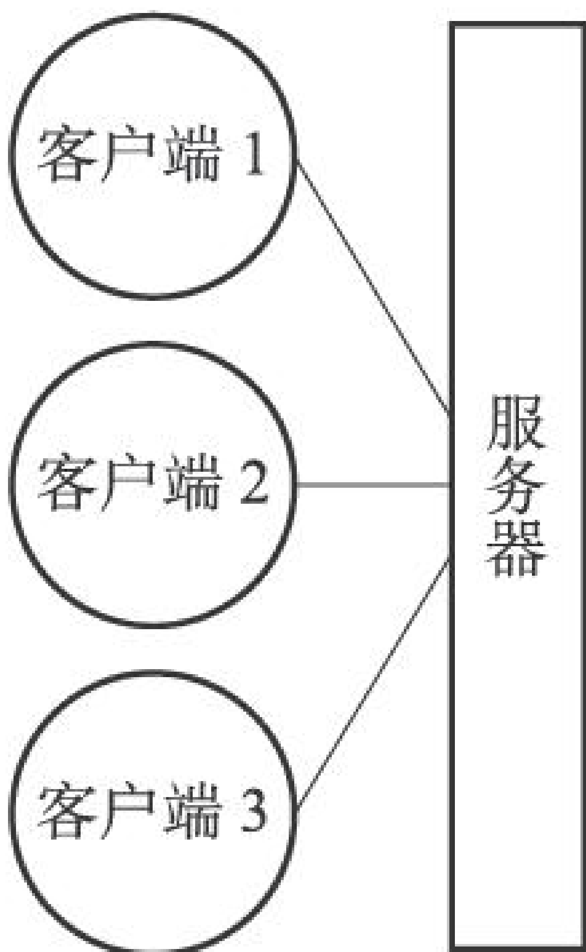


图13-1　客户端与服务器

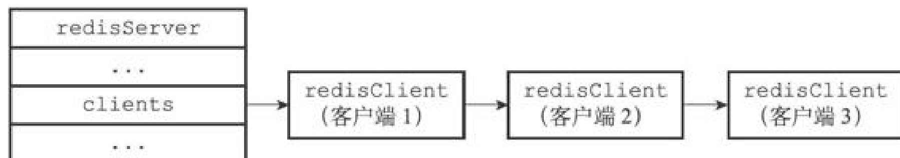


图13-2　clients链表

本章将对客户端状态的各个属性进行介绍，并讲述服务器创建并关闭各种不同类型的客户端的方法。

13.1 客户端属性

客户端状态包含的属性可以分为两类：

·一类是比较通用的属性，这些属性很少与特定功能相关，无论客户端执行的是什么工作，它们都要用到这些属性。

·另外一类是和特定功能相关的属性，比如操作数据库时需要用到的db属性和dictid属性，执行事务时需要用到的mstate属性，以及执行WATCH命令时需要用到的watched_keys属性等等。

本章将对客户端状态中比较通用的那部分属性进行介绍，至于那些和特定功能相关的属性，则会在相应的章节进行介绍。

13.1.1 套接字描述符

客户端状态的fd属性记录了客户端正在使用的套接字描述符：

```
typedef struct redisClient {
    // ...
    int fd;
    // ...
} redisClient;
```

根据客户端类型的不同，fd属性的值可以是-1或者是大于-1的整数：

·伪客户端（fake client）的fd属性的值为-1：伪客户端处理的命令请求来源于AOF文件或者Lua脚本，而不是网络，所以这种客户端不需要套接字连接，自然也不需要记录套接字描述符。目前Redis服务器会在两个地方用到伪客户端，一个用于载入AOF文件并还原数据库状态，而另一个则用于执行Lua脚本中包含的Redis命令。

·普通客户端的fd属性的值为大于-1的整数：普通客户端使用套接字来与服务器进行通信，所以服务器会用fd属性来记录客户端套接字的描述符。因为合法的套接字描述符不能是-1，所以普通客户端的套接字描述符的值必然是大于-1的整数。

执行CLIENT list命令可以列出目前所有连接到服务器的普通客户端，命令输出中的fd域显示了服务器连接客户端所使用的套接字描述符：

```
redis> CLIENT list
addr=127.0.0.1:53428 fd=6 name= age=1242 idle=0 ...
addr=127.0.0.1:53469 fd=7 name= age=4 idle=4 ...
```

13.1.2 名字

在默认情况下，一个连接到服务器的客户端是没有名字的。

比如在下面展示的CLIENT list命令示例中，两个客户端的name域都是空白的：

```
redis> CLIENT list
addr=127.0.0.1:53428 fd=6 name= age=1242 idle=0 ...
addr=127.0.0.1:53469 fd=7 name= age=4 idle=4 ...
```

使用CLIENT setname命令可以为客户端设置一个名字，让客户端的身份变得更清晰。

以下展示的是客户端执行CLIENT setname命令之后的客户端列表：

```
redis> CLIENT list
addr=127.0.0.1:53428 fd=6 name=message_queue age=2093 idle=0 ...
addr=127.0.0.1:53469 fd=7 name=user_relationship age=855 idle=2 ...
```

其中，第一个客户端的名字是message_queue，我们可以猜测它是负责处理消息队列的客户端；第二个客户端的名字是user_relationship，我们可以猜测它为负责处理用户关系的客户端。

客户端的名字记录在客户端状态的name属性里面：

```
typedef struct redisClient {
    // ...
    robj *name;
    // ...
} redisClient;
```

如果客户端没有为自己设置名字，那么相应客户端状态的name属性指向NULL指针；相反地，如果客户端为自己设置了名字，那么name属性将指向一个字符串对象，而该对象就保存着客户端的名字。

图13-3展示了一个客户端状态示例，根据name属性显示，客户端的名字为"message_queue"。

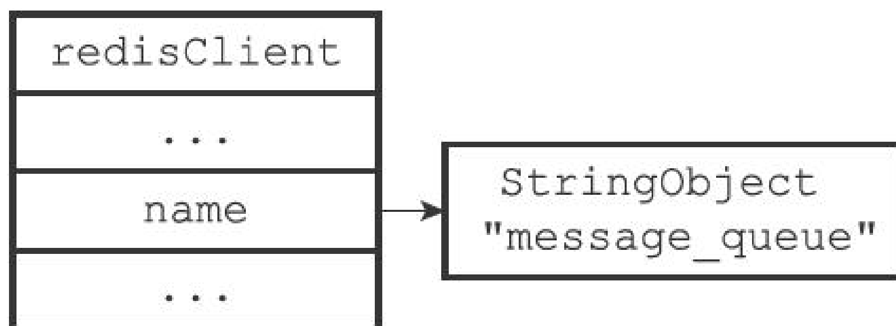


图13-3　name属性示例

13.1.3　标志

客户端的标志属性flags记录了客户端的角色（role），以及客户端目前所处的状态：

```
typedef struct redisClient {
    // ...
    int flags;
    // ...
} redisClient;
```

flags属性的值可以是单个标志：

```
flags = <flag>
```

也可以是多个标志的二进制或，比如：

```
flags = <flag1> | <flag2> | ...
```

每个标志使用一个常量表示，一部分标志记录了客户端的角色：

·在主从服务器进行复制操作时，主服务器会成为从服务器的客户端，而从服务器也会成为主服务器的客户端。REDIS_MASTER标志表示客户端代表的是一个主服务器，REDIS_SLAVE标志表示客户端代表的是一个从服务器。

·REDIS_PRE_PSYNC标志表示客户端代表的是一个版本低于Redis2.8的从服务器，主服务器不能使用PSYNC命令与这个从服务器进行同步。这个标志只能在REDIS_SLAVE标志处于打开状态时使用。

·REDIS_LUA_CLIENT标识表示客户端是专门用于处理Lua脚本里面包含的Redis命令的伪客户端。

而另外一部分标志则记录了客户端目前所处的状态：

·REDIS_MONITOR标志表示客户端正在执行MONITOR命令。

·REDIS_UNIX_SOCKET标志表示服务器使用UNIX套接字来连接客户端。

·REDIS_BLOCKED标志表示客户端正在被BRPOP、BLPOP等命令阻塞。

·REDIS_UNBLOCKED标志表示客户端已经从REDIS_BLOCKED标志所表示的阻塞状态中脱离出来，不再阻塞。REDIS_UNBLOCKED标志只

能在REDIS_BLOCKED标志已经打开的情况下使用。

·REDIS_MULTI标志表示客户端正在执行事务。

·REDIS_DIRTY_CAS标志表示事务使用WATCH命令监视的数据库键已经被修改，REDIS_DIRTY_EXEC标志表示事务在命令入队时出现了错误，以上两个标志都表示事务的安全性已经被破坏，只要这两个标记中的任意一个被打开，EXEC命令必然会执行失败。这两个标志只能在客户端打开了REDIS_MULTI标志的情况下使用。

·REDIS_CLOSE_ASAP标志表示客户端的输出缓冲区大小超出了服务器允许的范围，服务器会在下一次执行serverCron函数时关闭这个客户端，以免服务器的稳定性受到这个客户端影响。积存在输出缓冲区中的所有内容会直接被释放，不会返回给客户端。

·REDIS_CLOSE_AFTER_REPLY标志表示有用户对这个客户端执行了CLIENT KILL命令，或者客户端发送给服务器的命令请求中包含了错误的协议内容。服务器会将客户端积存在输出缓冲区中的所有内容发送给客户端，然后关闭客户端。

·REDIS_ASKING标志表示客户端向集群节点（运行在集群模式下的服务器）发送了ASKING命令。

·REDIS_FORCE_AOF标志强制服务器将当前执行的命令写入到AOF文件里面，REDIS_FORCE_REPL标志强制主服务器将当前执行的命令复制给所有从服务器。执行PUBSUB命令会使客户端打开REDIS_FORCE_AOF标志，执行SCRIPT LOAD命令会使客户端打开REDIS_FORCE_AOF标志和REDIS_FORCE_REPL标志。

·在主从服务器进行命令传播期间，从服务器需要向主服务器发送REPLICATION ACK命令，在发送这个命令之前，从服务器必须打开主服务器对应的客户端的REDIS_MASTER_FORCE_REPLY标志，否则发送操作会被拒绝执行。

以上提到的所有标志都定义在redis.h文件里面。

PUBSUB命令和SCRIPT LOAD命令的特殊性

通常情况下，Redis只会将那些对数据库进行了修改的命令写入到AOF文件，并复制到各个从服务器。如果一个命令没有对数据库进行任何修改，那么它就会被认为是只读命令，这个命令不会被写入到AOF文件，也不会被复制到从服务器。

以上规则适用于绝大部分Redis命令，但PUBSUB命令和SCRIPT

LOAD命令是其中的例外。PUBSUB命令虽然没有修改数据库，但PUBSUB命令向频道的所有订阅者发送消息这一行为带有副作用，接收到消息的所有客户端的状态都会因为这个命令而改变。因此，服务器需要使用REDIS_FORCE_AOF标志，强制将这个命令写入AOF文件，这样在将来载入AOF文件时，服务器就可以再次执行相同的PUBSUB命令，并产生相同的副作用。SCRIPT　LOAD命令的情况与PUBSUB命令类似：虽然SCRIPT　LOAD命令没有修改数据库，但它修改了服务器状态，所以它是一个带有副作用的命令，服务器需要使用REDIS_FORCE_AOF标志，强制将这个命令写入AOF文件，使得将来在载入AOF文件时，服务器可以产生相同的副作用。

另外，为了让主服务器和从服务器都可以正确地载入SCRIPT LOAD命令指定的脚本，服务器需要使用REDIS_FORCE_REPL标志，强制将SCRIPT LOAD命令复制给所有从服务器。

以下是一些flags属性的例子：

```
#
客户端是一个主服务器
REDIS_MASTER
#
客户端正在被列表命令阻塞
REDIS_BLOCKED
#
客户端正在执行事务，但事务的安全性已被破坏
REDIS_MULTI | REDIS_DIRTY_CAS
#
客户端是一个从服务器，并且版本低于Redis 2.8
REDIS_SLAVE | REDIS_PRE_PSYNC
#
这是专门用于执行Lua
脚本包含的Redis
命令的伪客户端
#
它强制服务器将当前执行的命令写入AOF
文件，并复制给从服务器
REDIS_LUA_CLIENT | REDIS_FORCE_AOF| REDIS_FORCE_REPL
```

13.1.4　输入缓冲区

客户端状态的输入缓冲区用于保存客户端发送的命令请求：

```
typedef struct redisClient {
    // ...
    sds querybuf;
    // ...
} redisClient;
```

举个例子，如果客户端向服务器发送了以下命令请求：

```
SET key value
```

那么客户端状态的querybuf属性将是一个包含以下内容的SDS值：

```
*3\r\n$3\r\nSET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n
```

图13-4展示了这个SDS值以及querybuf属性的样子。

输入缓冲区的大小会根据输入内容动态地缩小或者扩大，但它的最大大小不能超过1GB，否则服务器将关闭这个客户端。

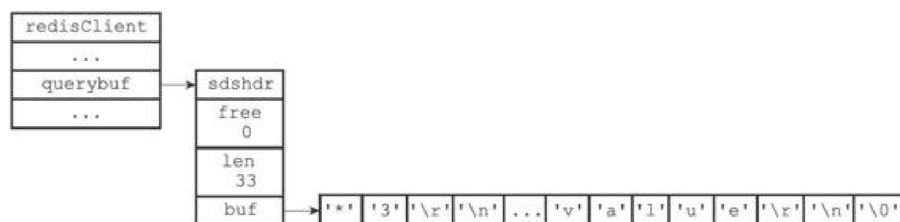


图13-4　querybuf属性示例

13.1.5　命令与命令参数

在服务器将客户端发送的命令请求保存到客户端状态的querybuf属性之后，服务器将对命令请求的内容进行分析，并将得出的命令参数以及命令参数的个数分别保存到客户端状态的argv属性和argc属性：

```
typedef struct redisClient {
    // ...
    robj **argv;
    int argc;
    // ...
} redisClient;
```

argv属性是一个数组，数组中的每个项都是一个字符串对象，其中argv[0]是要执行的命令，而之后的其他项则是传给命令的参数。

argc属性则负责记录argv数组的长度。

举个例子，对于图13-4所示的querybuf属性来说，服务器将分析并创建图13-5所示的argv属性和argc属性。

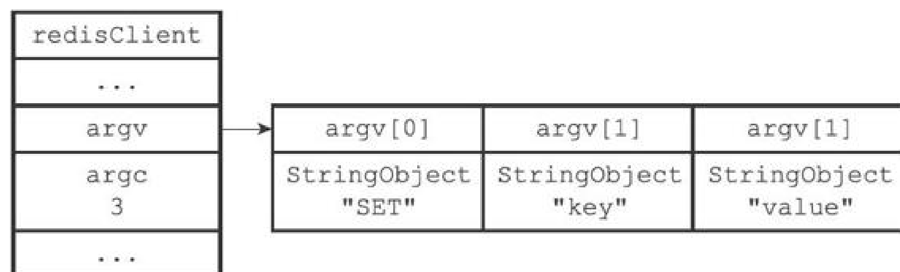


图13-5　argv属性和argc属性示例

注意，在图13-5展示的客户端状态中，argc属性的值为3，而不是2，因为命令的名字"SET"本身也是一个参数。

13.1.6 命令的实现函数

当服务器从协议内容中分析并得出argv属性和argc属性的值之后，服务器将根据项argv[0]的值，在命令表中查找命令所对应的命令实现函数。

图13-6展示了一个命令表示例，该表是一个字典，字典的键是一个SDS结构，保存了命令的名字，字典的值是命令所对应的redisCommand结构，这个结构保存了命令的实现函数、命令的标志、命令应该给定的参数个数、命令的总执行次数和总消耗时长等统计信息。

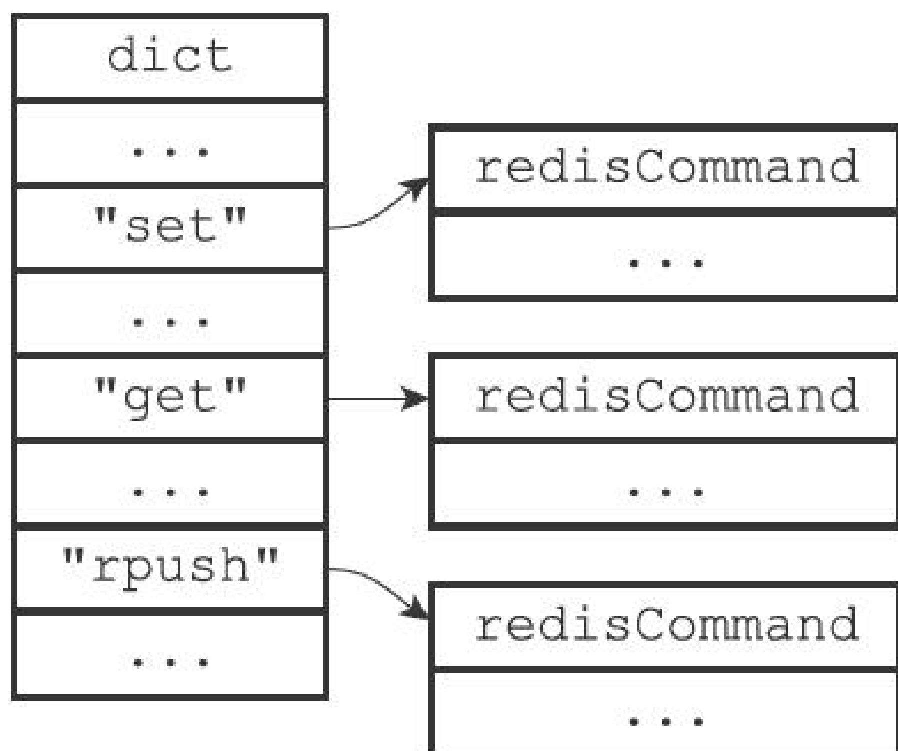


图13-6 命令表

当程序在命令表中成功找到argv[0]所对应的redisCommand结构时，它会将客户端状态的cmd指针指向这个结构：

```
typedef struct redisClient {
    // ...
    struct redisCommand *cmd;
    // ...
} redisClient;
```

之后，服务器就可以使用cmd属性所指向的redisCommand结构，以及argv、argc属性中保存的命令参数信息，调用命令实现函数，执行客户端指定的命令。

图13-7演示了服务器在argv[0]为"SET"时，查找命令表并将客户端状态的cmd指针指向目标redisCommand结构的整个过程。

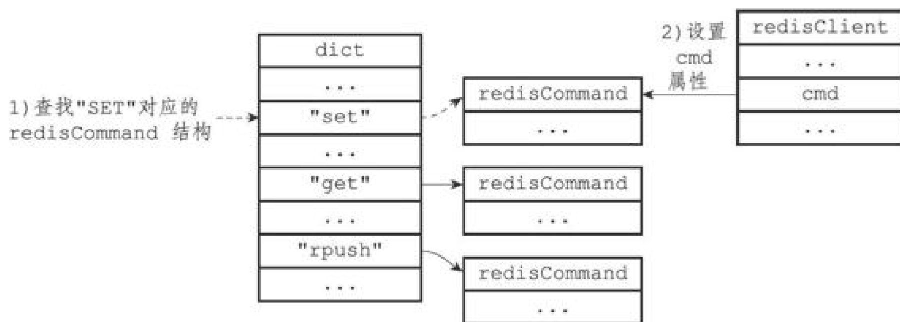


图13-7　查找命令并设置cmd属性

针对命令表的查找操作不区分输入字母的大小写，所以无论argv[0]是"SET"、"set"、或者"SeT"等等，查找的结果都是相同的。

13.1.7　输出缓冲区

执行命令所得的命令回复会被保存在客户端状态的输出缓冲区里面，每个客户端都有两个输出缓冲区可用，一个缓冲区的大小是固定的，另一个缓冲区的大小是可变的：

·固定大小的缓冲区用于保存那些长度比较小的回复，比如OK、简短的字符串值、整数值、错误回复等等。

·可变大小的缓冲区用于保存那些长度比较大的回复，比如一个非常长的字符串值，一个由很多项组成的列表，一个包含了很多元素的集合等等。

客户端的固定大小缓冲区由buf和bufpos两个属性组成：

```
typedef struct redisClient {
    // ...
    char buf[REDIS_REPLY_CHUNK_BYTES];
    int bufpos;
    // ...
} redisClient;
```

buf是一个大小为REDIS_REPLY_CHUNK_BYTES字节的字节数组，而bufpos属性则记录了buf数组目前已使用的字节数量。

REDIS_REPLY_CHUNK_BYTES常量目前的默认值为16*1024，也即

是说，buf数组的默认大小为16KB。

图13-8展示了一个使用固定大小缓冲区来保存返回值+OK\r\n的例子。

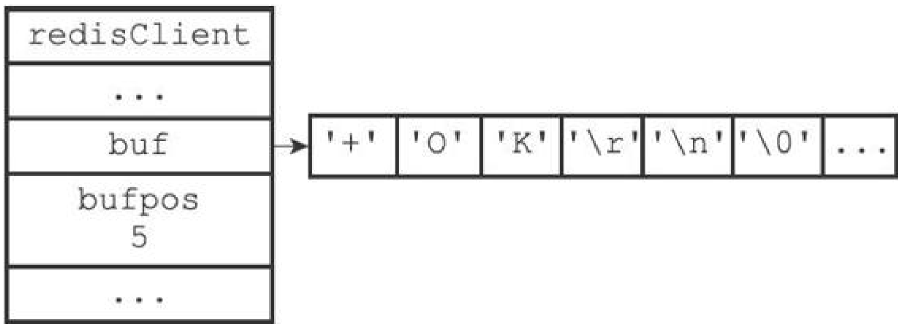


图13-8　固定大小缓冲区示例

当buf数组的空间已经用完，或者回复因为太大而没办法放进buf数组里面时，服务器就会开始使用可变大小缓冲区。

可变大小缓冲区由reply链表和一个或多个字符串对象组成：

```
typedef struct redisClient {
    // ...
    list *reply;
    // ...
} redisClient;
```

通过使用链表来连接多个字符串对象，服务器可以为客户端保存一个非常长的命令回复，而不必受到固定大小缓冲区16KB大小的限制。

图13-9展示了一个包含三个字符串对象的reply链表。

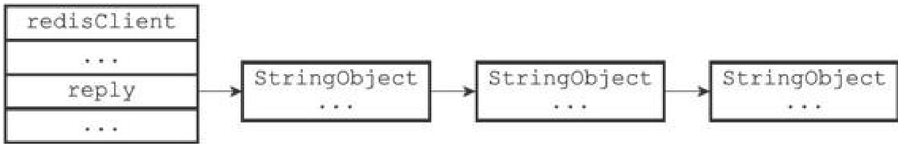


图13-9　可变大小缓冲区示例

13.1.8　身份验证

客户端状态的authenticated属性用于记录客户端是否通过了身份验证：

```
typedef struct redisClient {
    // ...
    int authenticated;
    // ...
} redisClient;
```

如果authenticated的值为0，那么表示客户端未通过身份验证；如果authenticated的值为1，那么表示客户端已经通过了身份验证。

举个例子，对于一个尚未进行身份验证的客户端来说，客户端状态的authenticated属性将如图13-10所示。



图13-10　未验证身份时的客户端状态

当客户端authenticated属性的值为0时，除了AUTH命令之外，客户端发送的所有其他命令都会被服务器拒绝执行：

```
redis> PING
(error) NOAUTH Authentication required.
redis> SET msg "hello world"
(error) NOAUTH Authentication required.
```

当客户端通过AUTH命令成功进行身份验证之后，客户端状态authenticated属性的值就会从0变为1，如图13-11所示，这时客户端就可以像往常一样向服务器发送命令请求了：

```
# authenticated
属性的值从0
变为1
redis> AUTH 123321
OK
redis> PING
PONG
redis> SET msg "hello world"
OK
```



图13-11　已经通过身份验证的客户端状态

authenticated属性仅在服务器启用了身份验证功能时使用。如果服务器没有启用身份验证功能的话，那么即使authenticated属性的值为0（这是默认值），服务器也不会拒绝执行客户端发送的命令请求。

关于服务器身份验证的更多信息可以参考示例配置文件对requirepass选项的相关说明。

13.1.9　时间

最后，客户端还有几个和时间有关的属性：

```
typedef struct redisClient {
    // ...
    time_t ctime;
    time_t lastinteraction;
    time_t obuf_soft_limit_reached_time;
    // ...
} redisClient;
```

ctime属性记录了创建客户端的时间，这个时间可以用来计算客户端与服务器已经连接了多少秒，CLIENT list命令的age域记录了这个秒数：

```
redis> CLIENT list
addr=127.0.0.1:53428 ... age=1242 ...
```

lastinteraction属性记录了客户端与服务器最后一次进行互动（interaction）的时间，这里的互动可以是客户端向服务器发送命令请求，也可以是服务器向客户端发送命令回复。

lastinteraction属性可以用来计算客户端的空转（idle）时间，也即

是，距离客户端与服务器最后一次进行互动以来，已经过去了多少秒，CLIENT list命令的idle域记录了这个秒数：

```
redis> CLIENT list
addr=127.0.0.1:53428 ... idle=12 ...
```

obuf_soft_limit_reached_time属性记录了输出缓冲区第一次到达软性限制（soft limit）的时间，稍后介绍输出缓冲区大小限制的时候会详细说明这个属性的作用。

13.2 客户端的创建与关闭

服务器使用不同的方式来创建和关闭不同类型的客户端，本节将介绍服务器创建和关闭客户端的方法。

13.2.1 创建普通客户端

如果客户端是通过网络连接与服务器进行连接的普通客户端，那么在客户端使用connect函数连接到服务器时，服务器就会调用连接事件处理器（在第12章有介绍），为客户端创建相应的客户端状态，并将这个新的客户端状态添加到服务器状态结构clients链表的末尾。

举个例子，假设当前有c1和c2两个普通客户端正在连接服务器，那么当一个新的普通客户端c3连接到服务器之后，服务器会将c3所对应的客户端状态添加到clients链表的末尾，如图13-12所示，其中用虚线包围的就是服务器为c3新创建的客户端状态。

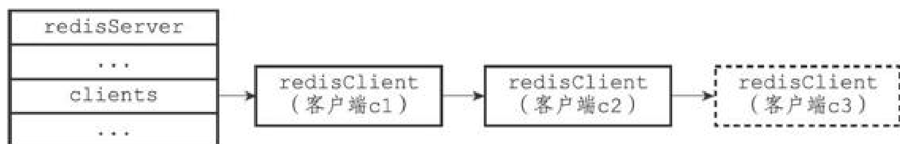


图13-12 服务器状态结构的clients链表

13.2.2 关闭普通客户端

一个普通客户端可以因为多种原因而被关闭：

·如果客户端进程退出或者被杀死，那么客户端与服务器之间的网络连接将被关闭，从而造成客户端被关闭。

·如果客户端向服务器发送了带有不符合协议格式的命令请求，那么这个客户端也会被服务器关闭。

·如果客户端成为了CLIENT KILL命令的目标，那么它也会被关闭。

·如果用户为服务器设置了timeout配置选项，那么当客户端的空转时间超过timeout选项设置的值时，客户端将被关闭。不过timeout选项有一些例外情况：如果客户端是主服务器（打开了REDIS_MASTER标志），从服务器（打开了REDIS_SLAVE标志），正在被BLPOP等命令阻塞（打开了REDIS_BLOCKED标志），或者正在执行SUBSCRIBE、PSUBSCRIBE等订阅命令，那么即使客户端的空转时间超过了timeout选项的值，客户端也不会被服务器关闭。

·如果客户端发送的命令请求的大小超过了输入缓冲区的限制大小

（默认为1 GB），那么这个客户端会被服务器关闭。

·如果要发送给客户端的命令回复的大小超过了输出缓冲区的限制大小，那么这个客户端会被服务器关闭。

前面介绍输出缓冲区的时候提到过，可变大小缓冲区由一个链表和任意多个字符串对象组成，理论上来说，这个缓冲区可以保存任意长的命令回复。

但是，为了避免客户端的回复过大，占用过多的服务器资源，服务器会时刻检查客户端的输出缓冲区的大小，并在缓冲区的大小超出范围时，执行相应的限制操作。

服务器使用两种模式来限制客户端输出缓冲区的大小：

·硬性限制（hard limit）：如果输出缓冲区的大小超过了硬性限制所设置的大小，那么服务器立即关闭客户端。

·软性限制（soft limit）：如果输出缓冲区的大小超过了软性限制所设置的大小，但还没超过硬性限制，那么服务器将使用客户端状态结构的obuf_soft_limit_reached_time属性记录下客户端到达软性限制的起始时间；之后服务器会继续监视客户端，如果输出缓冲区的大小一直超出软性限制，并且持续时间超过服务器设定的时长，那么服务器将关闭客户端；相反地，如果输出缓冲区的大小在指定时间之内，不再超出软性限制，那么客户端就不会被关闭，并且obuf_soft_limit_reached_time属性的值也会被清零。

使用client-output-buffer-limit选项可以为普通客户端、从服务器客户端、执行发布与订阅功能的客户端分别设置不同的软性限制和硬性限制，该选项的格式为：

```
client-output-buffer-limit <class> <hard limit> <soft limit> <soft seconds>
```

以下是三个设置示例：

```
client-output-buffer-limit normal 0 0 0
client-output-buffer-limit slave 256mb 64mb 60
client-output-buffer-limit pubsub 32mb 8mb 60
```

第一行设置将普通客户端的硬性限制和软性限制都设置为0，表示不限制客户端的输出缓冲区大小。

第二行设置将从服务器客户端的硬性限制设置为256MB，而软性限制设置为64MB，软性限制的时长为60秒。

第三行设置将执行发布与订阅功能的客户端的硬性限制设置为32MB，软性限制设置为8MB，软性限制的时长为60秒。

关于client-output-buffer-limit选项的更多用法，可以参考示例配置文件redis.conf。

13.2.3　Lua脚本的伪客户端

服务器会在初始化时创建负责执行Lua脚本中包含的Redis命令的伪客户端，并将这个伪客户端关联在服务器状态结构的lua_client属性中：

```
struct redisServer {
    // ...
    redisClient *lua_client;
    // ...
};
```

lua_client伪客户端在服务器运行的整个生命期中会一直存在，只有服务器被关闭时，这个客户端才会被关闭。

13.2.4　AOF文件的伪客户端

服务器在载入AOF文件时，会创建用于执行AOF文件包含的Redis命令的伪客户端，并在载入完成之后，关闭这个伪客户端。

13.3 重点回顾

·服务器状态结构使用clients链表连接起多个客户端状态，新添加的客户端状态会被放到链表的末尾。

·客户端状态的flags属性使用不同标志来表示客户端的角色，以及客户端当前所处的状态。

·输入缓冲区记录了客户端发送的命令请求，这个缓冲区的大小不能超过1GB。

·命令的参数和参数个数会被记录在客户端状态的argv和argc属性里面，而cmd属性则记录了客户端要执行命令的实现函数。

·客户端有固定大小缓冲区和可变大小缓冲区两种缓冲区可用，其中固定大小缓冲区的最大大小为16KB，而可变大小缓冲区的最大大小不能超过服务器设置的硬性限制值。

·输出缓冲区限制值有两种，如果输出缓冲区的大小超过了服务器设置的硬性限制，那么客户端会被立即关闭；除此之外，如果客户端在一定时间内，一直超过服务器设置的软性限制，那么客户端也会被关闭。

·当一个客户端通过网络连接连上服务器时，服务器会为这个客户端创建相应的客户端状态。网络连接关闭、发送了不合协议格式的命令请求、成为CLIENT KILL命令的目标、空转时间超时、输出缓冲区的大小超出限制，以上这些原因都会造成客户端被关闭。

·处理Lua脚本的伪客户端在服务器初始化时创建，这个客户端会一直存在，直到服务器关闭。

·载入AOF文件时使用的伪客户端在载入工作开始时动态创建，载入工作完毕之后关闭。

第14章　服务器

Redis服务器负责与多个客户端建立网络连接，处理客户端发送的命令请求，在数据库中保存客户端执行命令所产生的数据，并通过资源管理来维持服务器自身的运转。

本章的第一节将以服务器执行SET命令的过程作为例子，展示服务器处理命令请求的整个过程，说明在执行命令的过程中，服务器和客户端进行了什么交互，服务器中的各个不同组件又是如何协作的，等等。

本章的第二节将对serverCron函数进行介绍，详细列举这个函数执行的操作，并说明这些操作对于服务器维持正常运行有何帮助。

本章的最后一节将对服务器的启动过程进行介绍，通过了解Redis服务器的启动过程可以知道，在启动服务器程序、直到服务器可以接受客户端命令请求的这段时间里，服务器都做了些什么准备工作。

14.1　命令请求的执行过程

一个命令请求从发送到获得回复的过程中，客户端和服务器需要完成一系列操作。举个例子，如果我们使用客户端执行以下命令：

```
redis> SET KEY VALUE
OK
```

那么从客户端发送SET KEY VALUE命令到获得回复OK期间，客户端和服务器共需要执行以下操作：

1）客户端向服务器发送命令请求SET KEY VALUE。

2）服务器接收并处理客户端发来的命令请求SET KEY VALUE，在数据库中进行设置操作，并产生命令回复OK。

3）服务器将命令回复OK发送给客户端。

4）客户端接收服务器返回的命令回复OK，并将这个回复打印给用户观看。

本节接下来的内容将对这些操作的执行细节进行补充，详细地说明客户端和服务器在执行命令请求时所做的各种工作。

14.1.1　发送命令请求

Redis服务器的命令请求来自Redis客户端，当用户在客户端中键入一个命令请求时，客户端会将这个命令请求转换成协议格式，然后通过连接到服务器的套接字，将协议格式的命令请求发送给服务器，如图14-1所示。



图14-1　客户端接收并发送命令请求的过程

举个例子，假设用户在客户端键入了命令：

```
SET KEY VALUE
```

那么客户端会将这个命令转换成协议：

```
*3\r\n$3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n
```

然后将这段协议内容发送给服务器。

14.1.2　读取命令请求

当客户端与服务器之间的连接套接字因为客户端的写入而变得可读时，服务器将调用命令请求处理器来执行以下操作：

1）读取套接字中协议格式的命令请求，并将其保存到客户端状态的输入缓冲区里面。

2）对输入缓冲区中的命令请求进行分析，提取出命令请求中包含的命令参数，以及命令参数的个数，然后分别将参数和参数个数保存到客户端状态的argv属性和argc属性里面。

3）调用命令执行器，执行客户端指定的命令。

继续用上一个小节的SET命令为例子，图14-2展示了程序将命令请求保存到客户端状态的输入缓冲区之后，客户端状态的样子。

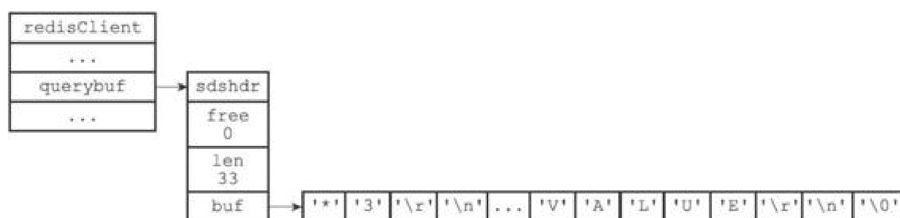


图14-2　客户端状态中的命令请求

之后，分析程序将对输入缓冲区中的协议进行分析：

```
*3\r\n$3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n
```

并将得出的分析结果保存到客户端状态的argv属性和argc属性里面，如图14-3所示。

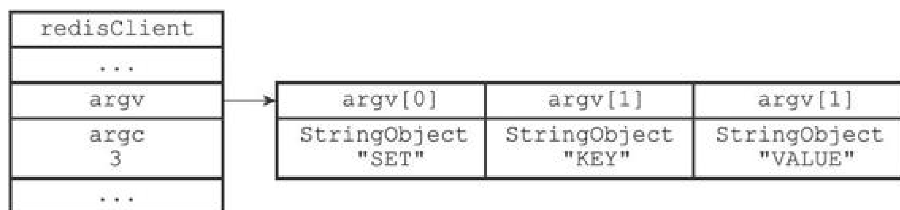


图14-3　客户端状态的argv属性和argc属性

之后，服务器将通过调用命令执行器来完成执行命令所需的余下步骤，以下几个小节将分别介绍命令执行器所执行的工作。

14.1.3　命令执行器（1）：查找命令实现

命令执行器要做的第一件事就是根据客户端状态的argv[0]参数，在命令表（command　table）中查找参数所指定的命令，并将找到的命令保存到客户端状态的cmd属性里面。

命令表是一个字典，字典的键是一个个命令名字，比如"set"、"get"、"del"等等；而字典的值则是一个个redisCommand结构，每个redisCommand结构记录了一个Redis命令的实现信息，表14-1记录了这个结构的各个主要属性的类型和作用。

表14-1　redisCommand结构的主要属性

属性名	类型	作用
name	char *	命令的名字，比如"set"
proc	redisCommandProc *	函数指针，指向命令的实现函数，比如setCommand。redisCommandProc类型的定义为typedef void redisCommandProc(redisClient *c);
arity	int	命令参数的个数，用于检查命令请求的格式是否正确。如果这个值为负数-N，那么表示参数的数量大于等于N。注意命令的名字本身也是一个参数，比如说SET msg "hello world"命令的参数是"SET"、"msg"、"hello world"，而不仅仅是"msg"和"hello world"
sflags	char *	字符串形式的标识值，这个值记录了命令的属性，比如这个命令是写命令还是读命令，这个命令是否允许在载入数据时使用，这个命令是否允许在Lua脚本中使用等等
flags	int	对sflags标识进行分析得出的二进制标识，由程序自动生成。服务器对命令标识进行检查时使用的都是flags属性而不是sflags属性，因为对二进制标识的检查可以方便地通过&、^、~等操作来完成
calls	long long	服务器总共执行了多少次这个命令
milliseconds	long long	服务器执行这个命令所耗费的总时长

表14-2列出了sflags属性可以使用的标识值，以及这些标识的意义。

表14-2　sflags属性的标识

标识	意义	带有这个标识的命令
w	这是一个写入命令，可能会修改数据库	SET、RPUSH、DEL 等等
r	这是一个只读命令，不会修改数据库	GET、STRLEN、EXISTS 等等
m	这个命令可能会占用大量内存，执行之前需要先检查服务器的内存使用情况，如果内存紧缺的话就禁止执行这个命令	SET、APPEND、RPUSH、LPUSH、SADD、SINTERSTORE 等等
a	这是一个管理命令	SAVE、BGSAVE、SHUTDOWN 等等
p	这是一个发布与订阅功能方面的命令	PUBLISH、SUBSCRIBE、PUBSUB 等等
s	这个命令不可以在 Lua 脚本中使用	BRPOP、BLPOP、BRPOPLPUSH、SPOP 等等
R	这是一个随机命令，对于相同的数据集和相同的参数，命令返回的结果可能不同	SPOP、SRANDMEMBER、SSCAN、RANDOMKEY 等等
S	当在 Lua 脚本中使用这个命令时，对这个命令的输出结果进行一次排序，使得命令的结果有序	SINTER、SUNION、SDIFF、SMEMBERS、KEYS 等等
l	这个命令可以在服务器载入数据的过程中使用	INFO、SHUTDOWN、PUBLISH 等等
t	这是一个允许从服务器在带有过期数据时使用的命令	SLAVEOF、PING、INFO 等等
M	这个命令在监视器（monitor）模式下不会自动被传播（propagate）	EXEC

图14-4展示了命令表的样子，并且以SET命令和GET命令作为例子，展示了redisCommand结构：

·SET命令的名字为"set"，实现函数为setCommand；命令的参数个数为-3，表示命令接受三个或以上数量的参数；命令的标识为"wm"，表示SET命令是一个写入命令，并且在执行这个命令之前，服务器应该对占用内存状况进行检查，因为这个命令可能会占用大量内存。

·GET命令的名字为"get"，实现函数为getCommand函数；命令的参数个数为2，表示命令只接受两个参数；命令的标识为"r"，表示这是一个只读命令。

继续之前SET命令的例子，当程序以图14-3中的argv[0]作为输入，在命令表中进行查找时，命令表将返回"set"键所对应的redisCommand结构，客户端状态的cmd指针会指向这个redisCommand结构，如图14-5所示。

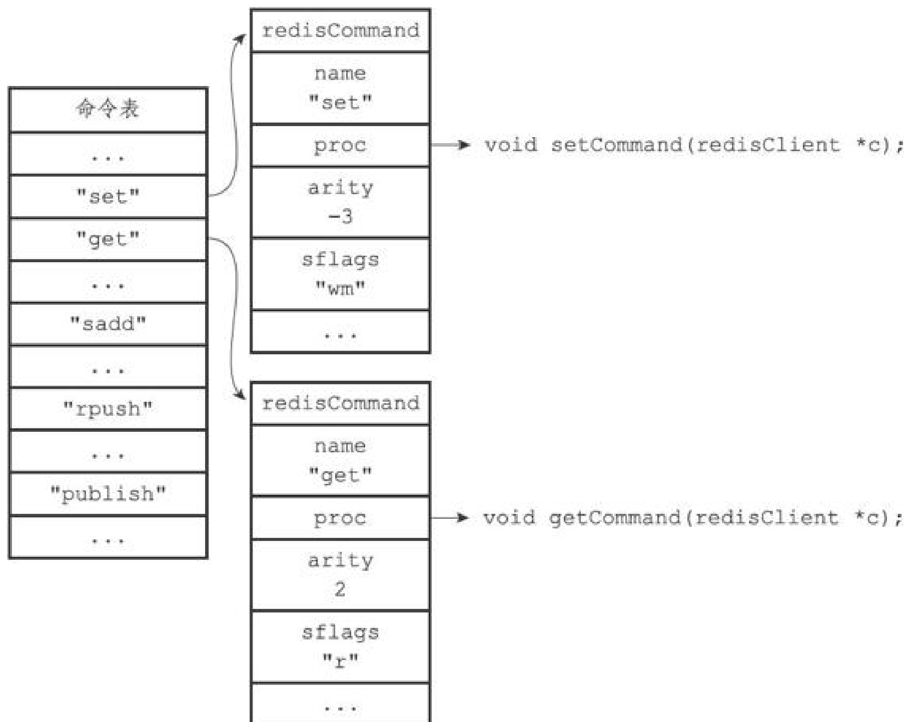


图14-4　命令表

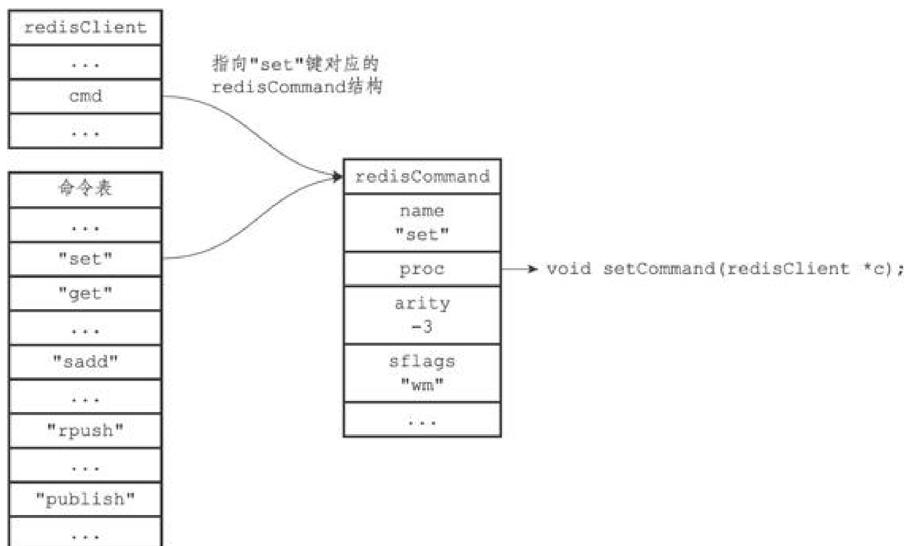


图14-5　设置客户端状态的cmd指针

命令名字的大小写不影响命令表的查找结果

因为命令表使用的是大小写无关的查找算法，无论输入的命令

名字是大写、小写或者混合大小写，只要命令的名字是正确的，就能找到相应的redisCommand结构。比如说，无论用户输入的命令名字是"SET"、"set"、"SeT"又或者"sEt"，命令表返回的都是同一个redisCommand结构。这也是Redis客户端可以发送不同大小写的命令，并且获得相同执行结果的原因：

```
#
以下四个命令的执行效果完全一样
redis> SET msg "hello world"
OK
redis> set msg "hello world"
OK
redis> SeT msg "hello world"
OK
redis> sEt msg "hello world"
OK
```

14.1.4　命令执行器（2）：执行预备操作

到目前为止，服务器已经将执行命令所需的命令实现函数（保存在客户端状态的cmd属性）、参数（保存在客户端状态的argv属性）、参数个数（保存在客户端状态的argc属性）都收集齐了，但是在真正执行命令之前，程序还需要进行一些预备操作，从而确保命令可以正确、顺利地被执行，这些操作包括：

·检查客户端状态的cmd指针是否指向NULL，如果是的话，那么说明用户输入的命令名字找不到相应的命令实现，服务器不再执行后续步骤，并向客户端返回一个错误。

·根据客户端cmd属性指向的redisCommand结构的arity属性，检查命令请求所给定的参数个数是否正确，当参数个数不正确时，不再执行后续步骤，直接向客户端返回一个错误。比如说，如果redisCommand结构的arity属性的值为-3，那么用户输入的命令参数个数必须大于等于3个才行。

·检查客户端是否已经通过了身份验证，未通过身份验证的客户端只能执行AUTH命令，如果未通过身份验证的客户端试图执行除AUTH命令之外的其他命令，那么服务器将向客户端返回一个错误。

·如果服务器打开了maxmemory功能，那么在执行命令之前，先检查服务器的内存占用情况，并在有需要时进行内存回收，从而使得接下来的命令可以顺利执行。如果内存回收失败，那么不再执行后续步骤，向客户端返回一个错误。

·如果服务器上一次执行BGSAVE命令时出错，并且服务器打开了stop-writes-on-bgsave-error功能，而且服务器即将要执行的命令是一个写命令，那么服务器将拒绝执行这个命令，并向客户端返回一个错误。

·如果客户端当前正在用SUBSCRIBE命令订阅频道，或者正在用PSUBSCRIBE命令订阅模式，那么服务器只会执行客户端发来的SUBSCRIBE、PSUBSCRIBE、UNSUBSCRIBE、PUNSUBSCRIBE四个命令，其他命令都会被服务器拒绝。

·如果服务器正在进行数据载入，那么客户端发送的命令必须带有l标识（比如INFO、SHUTDOWN、PUBLISH等等）才会被服务器执行，其他命令都会被服务器拒绝。

·如果服务器因为执行Lua脚本而超时并进入阻塞状态，那么服务器只会执行客户端发来的SHUTDOWN nosave命令和SCRIPT KILL命令，其他命令都会被服务器拒绝。

·如果客户端正在执行事务，那么服务器只会执行客户端发来的EXEC、DISCARD、MULTI、WATCH四个命令，其他命令都会被放进事务队列中。

·如果服务器打开了监视器功能，那么服务器会将要执行的命令和参数等信息发送给监视器。当完成了以上预备操作之后，服务器就可以开始真正执行命令了。


注意

以上只列出了服务器在单机模式下执行命令时的检查操作，当服务器在复制或者集群模式下执行命令时，预备操作还会更多一些。

14.1.5　命令执行器（3）：调用命令的实现函数

在前面的操作中，服务器已经将要执行命令的实现保存到了客户端状态的cmd属性里面，并将命令的参数和参数个数分别保存到了客户端状态的argv属性和argv属性里面，当服务器决定要执行命令时，它只要执行以下语句就可以了：

```
// client
是指向客户端状态的指针
client->cmd->proc(client);
```

因为执行命令所需的实际参数都已经保存到客户端状态的argv属性里面了，所以命令的实现函数只需要一个指向客户端状态的指针作为参数即可。

继续以之前的SET命令为例子，图14-6展示了客户端包含了命令实

现、参数和参数个数的样子。

对于这个例子来说，执行语句：

```
client->cmd->proc(client);
```

等于执行语句：

```
setCommand(client);
```

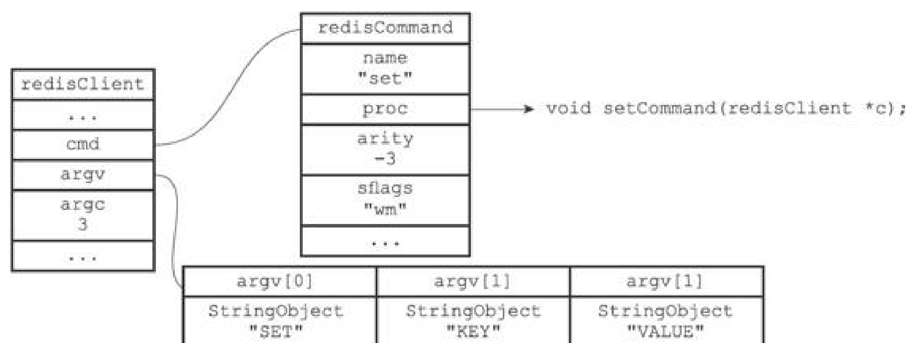


图14-6　客户端状态

被调用的命令实现函数会执行指定的操作，并产生相应的命令回复，这些回复会被保存在客户端状态的输出缓冲区里面（buf属性和reply属性），之后实现函数还会为客户端的套接字关联命令回复处理器，这个处理器负责将命令回复返回给客户端。

对于前面SET命令的例子来说，函数调用setCommand（client）将产生一个"+OK\r\n"回复，这个回复会被保存到客户端状态的buf属性里面，如图14-7所示。

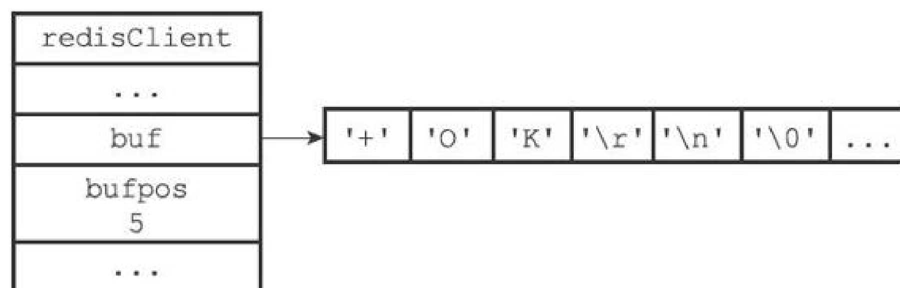


图14-7　保存了命令回复的客户端状态

14.1.6　命令执行器（4）：执行后续工作

在执行完实现函数之后，服务器还需要执行一些后续工作：

·如果服务器开启了慢查询日志功能，那么慢查询日志模块会检查是否需要为刚刚执行完的命令请求添加一条新的慢查询日志。

·根据刚刚执行命令所耗费的时长，更新被执行命令的redisCommand结构的milliseconds属性，并将命令的redisCommand结构的calls计数器的值增一。

·如果服务器开启了AOF持久化功能，那么AOF持久化模块会将刚刚执行的命令请求写入到AOF缓冲区里面。

·如果有其他从服务器正在复制当前这个服务器，那么服务器会将刚刚执行的命令传播给所有从服务器。

当以上操作都执行完了之后，服务器对于当前命令的执行到此就告一段落了，之后服务器就可以继续从文件事件处理器中取出并处理下一个命令请求了。

14.1.7　将命令回复发送给客户端

前面说过，命令实现函数会将命令回复保存到客户端的输出缓冲区里面，并为客户端的套接字关联命令回复处理器，当客户端套接字变为可写状态时，服务器就会执行命令回复处理器，将保存在客户端输出缓冲区中的命令回复发送给客户端。

当命令回复发送完毕之后，回复处理器会清空客户端状态的输出缓冲区，为处理下一个命令请求做好准备。

以图14-7所示的客户端状态为例子，当客户端的套接字变为可写状态时，命令回复处理器会将协议格式的命令回复"+OK\r\n"发送给客户端。

14.1.8　客户端接收并打印命令回复

当客户端接收到协议格式的命令回复之后，它会将这些回复转换成人类可读的格式，并打印给用户观看（假设我们使用的是Redis自带的redis-cli客户端），如图14-8所示。



图14-8　客户端接收并打印命令回复的过程

继续以之前的SET命令为例子，当客户端接到服务器发来的"+OK\r\n"协议回复时，它会将这个回复转换成"OK\n"，然后打印给用户看：

```
redis> SET KEY VALUE
OK
```

以上就是Redis客户端和服务器执行命令请求的整个过程了。

14.2 serverCron函数

Redis服务器中的serverCron函数默认每隔100毫秒执行一次，这个函数负责管理服务器的资源，并保持服务器自身的良好运转。

本节接下来的内容将对serverCron函数执行的操作进行完整介绍，并介绍redisServer结构（服务器状态）中和serverCron函数有关的属性。

14.2.1 更新服务器时间缓存

Redis服务器中有不少功能需要获取系统的当前时间，而每次获取系统的当前时间都需要执行一次系统调用，为了减少系统调用的执行次数，服务器状态中的unixtime属性和mstime属性被用作当前时间的缓存：

```
struct redisServer {
    // ...
    //
保存了秒级精度的系统当前UNIX
时间戳
    time_t unixtime;
    //
保存了毫秒级精度的系统当前UNIX
时间戳
    long long mstime;
    // ...
};
```

因为serverCron函数默认会以每100毫秒一次的频率更新unixtime属性和mstime属性，所以这两个属性记录的时间的精确度并不高：

·服务器只会在打印日志、更新服务器的LRU时钟、决定是否执行持久化任务、计算服务器上线时间（uptime）这类对时间精确度要求不高的功能上。

·对于为键设置过期时间、添加慢查询日志这种需要高精确度时间的功能来说，服务器还是会再次执行系统调用，从而获得最准确的系统当前时间。

14.2.2 更新LRU时钟

服务器状态中的lruclock属性保存了服务器的LRU时钟，这个属性和上面介绍的unixtime属性、mstime属性一样，都是服务器时间缓存的一种：

```
struct redisServer {
    // ...
    //
```

```
默认每10
秒更新一次的时钟缓存，
    //
用于计算键的空转（idle
）时长。
    unsigned lruclock:22;
    // ...
};
```

每个Redis对象都会有一个lru属性，这个lru属性保存了对象最后一次被命令访问的时间：

```
typedef struct redisObject {
    // ...
    unsigned lru:22;
    // ...
} robj;
```

当服务器要计算一个数据库键的空转时间（也即是数据库键对应的值对象的空转时间），程序会用服务器的lruclock属性记录的时间减去对象的lru属性记录的时间，得出的计算结果就是这个对象的空转时间：

```
redis> SET msg "hello world"
OK
#
等待一小段时间
redis> OBJECT IDLETIME msg
(integer)20
#
等待一阵子
redis> OBJECT IDLETIME msg
(integer)180
#
访问msg
键的值
redis> GET msg
"hello world"
#
键处于活跃状态，空转时长为0
redis> OBJECT IDLETIME msg
(integer)0
```

serverCron函数默认会以每10秒一次的频率更新lruclock属性的值，因为这个时钟不是实时的，所以根据这个属性计算出来的LRU时间实际上只是一个模糊的估算值。

lruclock时钟的当前值可以通过INFO server命令的lru_clock域查看：

```
redis> INFO server
# Server
...
lru_clock:55923
...
```

14.2.3 更新服务器每秒执行命令次数

serverCron函数中的trackOperationsPerSecond函数会以每100毫秒一次的频率执行，这个函数的功能是以抽样计算的方式，估算并记录服务器在最近一秒钟处理的命令请求数量，这个值可以通过INFO status命令的instantaneous_ops_per_sec域查看：

```
redis> INFO stats
# Stats
...
instantaneous_ops_per_sec:6
...
```

上面的命令结果显示，在最近的一秒钟内，服务器处理了大概六个命令。

trackOperationsPerSecond函数和服务器状态中四个ops_sec_开头的属性有关：

```
struct redisServer {
    // ...
    //
上一次进行抽样的时间
    long long ops_sec_last_sample_time;
    //
上一次抽样时，服务器已执行命令的数量
    long long ops_sec_last_sample_ops;
    // REDIS_OPS_SEC_SAMPLES
大小（默认值为16
）的环形数组，
    //
数组中的每个项都记录了一次抽样结果。
    long long ops_sec_samples[REDIS_OPS_SEC_SAMPLES];
    // ops_sec_samples
数组的索引值，
    //
每次抽样后将值自增一，
    //
在值等于16
时重置为0
，
    //
让ops_sec_samples
数组构成一个环形数组。
    int ops_sec_idx;
    // ...
};
```

trackOperationsPerSecond函数每次运行，都会根据ops_sec_last_sample_time记录的上一次抽样时间和服务器的当前时间，以及ops_sec_last_sample_ops记录的上一次抽样的已执行命令数量和服务器当前的已执行命令数量，计算出两次trackOperationsPerSecond调用之间，服务器平均每一毫秒处理了多少个命令请求，然后将这个平均值乘以1000，这就得到了服务器在一秒钟内能处理多少个命令请求的估计值，这个估计值会被作为一个新的数组项被放进ops_sec_samples环形数组里面。

当客户端执行INFO命令时，服务器就会调用

getOperationsPerSecond函数，根据ops_sec_samples环形数组中的抽样结果，计算出instantaneous_ops_per_sec属性的值，以下是getOperationsPerSecond函数的实现代码：

```
long long getOperationsPerSecond(void){
    int j;
    long long sum = 0;
    //
计算所有取样值的总和
    for (j = 0; j < REDIS_OPS_SEC_SAMPLES; j++)
        sum += server.ops_sec_samples[j];
    //
计算取样的平均值
    return sum / REDIS_OPS_SEC_SAMPLES;
}
```

根据getOperationsPerSecond函数的定义可以看出，instantaneous_ops_per_sec属性的值是通过计算最近REDIS_OPS_SEC_SAMPLES次取样的平均值来计算得出的，它只是一个估算值。

14.2.4 更新服务器内存峰值记录

服务器状态中的stat_peak_memory属性记录了服务器的内存峰值大小：

```
struct redisServer {
    // ...
    //
已使用内存峰值
    size_t stat_peak_memory;
    // ...
};
```

每次serverCron函数执行时，程序都会查看服务器当前使用的内存数量，并与stat_peak_memory保存的数值进行比较，如果当前使用的内存数量比stat_peak_memory属性记录的值要大，那么程序就将当前使用的内存数量记录到stat_peak_memory属性里面。

INFO memory命令的used_memory_peak和used_memory_peak_human两个域分别以两种格式记录了服务器的内存峰值：

```
redis> INFO memory
# Memory
...
used_memory_peak:501824
used_memory_peak_human:490.06K
...
```

14.2.5 处理SIGTERM信号

在启动服务器时，Redis会为服务器进程的SIGTERM信号关联处理器sigtermHandler函数，这个信号处理器负责在服务器接到SIGTERM信号时，打开服务器状态的shutdown_asap标识：

```
// SIGTERM
信号的处理器
static void sigtermHandler(int sig) {
    //
打印日志
    redisLogFromHandler(REDIS_WARNING,"Received SIGTERM, scheduling shutdown...");
    //
打开关闭标识
    server.shutdown_asap = 1;
}
```

每次serverCron函数运行时，程序都会对服务器状态的shutdown_asap属性进行检查，并根据属性的值决定是否关闭服务器：

```
struct redisServer {
    // ...
    //
关闭服务器的标识：
    //
值为1
时，关闭服务器，
    //
值为0
时，不做动作。
    int shutdown_asap;
    // ...
};
```

以下代码展示了服务器在接到SIGTERM信号之后，关闭服务器并打印相关日志的过程：

```
[6794 | signal handler] (1384435690) Received SIGTERM, scheduling shutdown...
[6794] 14 Nov 21:28:10.108 # User requested shutdown...
[6794] 14 Nov 21:28:10.108 * Saving the final RDB snapshot before exiting.
[6794] 14 Nov 21:28:10.161 * DB saved on disk
[6794] 14 Nov 21:28:10.161 # Redisis now ready to exit, bye bye...
```

从日志里面可以看到，服务器在关闭自身之前会进行RDB持久化操作，这也是服务器拦截SIGTERM信号的原因，如果服务器一接到SIGTERM信号就立即关闭，那么它就没办法执行持久化操作了。

14.2.6　管理客户端资源

serverCron函数每次执行都会调用clientsCron函数，clientsCron函数会对一定数量的客户端进行以下两个检查：

·如果客户端与服务器之间的连接已经超时（很长一段时间里客户端和服务器都没有互动），那么程序释放这个客户端。

·如果客户端在上一次执行命令请求之后，输入缓冲区的大小超过了

一定的长度，那么程序会释放客户端当前的输入缓冲区，并重新创建一个默认大小的输入缓冲区，从而防止客户端的输入缓冲区耗费了过多的内存。

14.2.7 管理数据库资源

serverCron函数每次执行都会调用databasesCron函数，这个函数会对服务器中的一部分数据库进行检查，删除其中的过期键，并在有需要时，对字典进行收缩操作，第9章经对这些操作进行了详细的说明。

14.2.8 执行被延迟的BGREWRITEAOF

在服务器执行BGSAVE命令的期间，如果客户端向服务器发来BGREWRITEAOF命令，那么服务器会将BGREWRITEAOF命令的执行时间延迟到BGSAVE命令执行完毕之后。

服务器的aof_rewrite_scheduled标识记录了服务器是否延迟了BGREWRITEAOF命令：

```
struct redisServer {
    // ...
    //
如果值为1
，那么表示有 BGREWRITEAOF
命令被延迟了。
    int aof_rewrite_scheduled;
    // ...
};
```

每次serverCron函数执行时，函数都会检查BGSAVE命令或者BGREWRITEAOF命令是否正在执行，如果这两个命令都没在执行，并且aof_rewrite_scheduled属性的值为1，那么服务器就会执行之前被推延的BGREWRITEAOF命令。

14.2.9 检查持久化操作的运行状态

服务器状态使用rdb_child_pid属性和aof_child_pid属性记录执行BGSAVE命令和BGREWRITEAOF命令的子进程的ID，这两个属性也可以用于检查BGSAVE命令或者BGREWRITEAOF命令是否正在执行：

```
struct redisServer {
    // ...
    //
记录执行BGSAVE
命令的子进程的ID
：
    //
如果服务器没有在执行BGSAVE
，
    //
那么这个属性的值为-1
。
```

```
    pid_t rdb_child_pid;                /* PID of RDB saving child */
    //
记录执行BGREWRITEAOF
命令的子进程的ID
：
    //
如果服务器没有在执行BGREWRITEAOF
，
    //
那么这个属性的值为-1
。
    pid_t aof_child_pid;                /* PID if rewriting process */
    // ...
};
```

每次serverCron函数执行时，程序都会检查rdb_child_pid和aof_child_pid两个属性的值，只要其中一个属性的值不为-1，程序就会执行一次wait3函数，检查子进程是否有信号发来服务器进程：

·如果有信号到达，那么表示新的RDB文件已经生成完毕（对于BGSAVE命令来说），或者AOF文件已经重写完毕（对于BGREWRITEAOF命令来说），服务器需要进行相应命令的后续操作，比如用新的RDB文件替换现有的RDB文件，或者用重写后的AOF文件替换现有的AOF文件。

·如果没有信号到达，那么表示持久化操作未完成，程序不做动作。

另一方面，如果rdb_child_pid和aof_child_pid两个属性的值都为-1，那么表示服务器没有在进行持久化操作，在这种情况下，程序执行以下三个检查：

1）查看是否有BGREWRITEAOF被延迟了，如果有的话，那么开始一次新的BGREWRITEAOF操作（这就是上一个小节我们说到的检查）。

2）检查服务器的自动保存条件是否已经被满足，如果条件满足，并且服务器没有在执行其他持久化操作，那么服务器开始一次新的BGSAVE操作（因为条件1可能会引发一次BGREWRITEAOF，所以在这个检查中，程序会再次确认服务器是否已经在执行持久化操作了）。

3）检查服务器设置的AOF重写条件是否满足，如果条件满足，并且服务器没有在执行其他持久化操作，那么服务器将开始一次新的BGREWRITEAOF操作（因为条件1和条件2都可能会引起新的持久化操作，所以在这个检查中，我们要再次确认服务器是否已经在执行持久化操作了）。

图14-9以流程图的方式展示了这个检查过程。

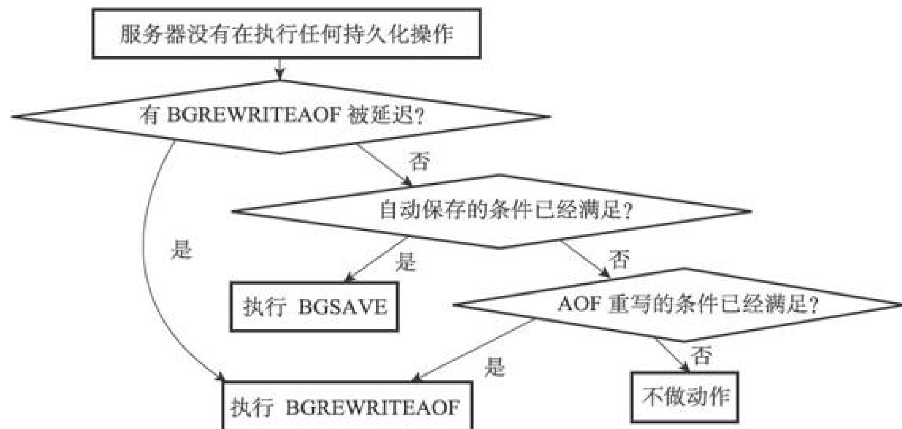


图14-9　判断是否需要执行持久化操作

14.2.10　将AOF缓冲区中的内容写入AOF文件

如果服务器开启了AOF持久化功能，并且AOF缓冲区里面还有待写入的数据，那么serverCron函数会调用相应的程序，将AOF缓冲区中的内容写入到AOF文件里面，第11章对此有详细的说明。

14.2.11　关闭异步客户端

在这一步，服务器会关闭那些输出缓冲区大小超出限制的客户端，第13章对此有详细的说明。

14.2.12　增加cronloops计数器的值

服务器状态的cronloops属性记录了serverCron函数执行的次数：

```
struct redisServer {
    // ...
    // serverCron
函数的运行次数计数器
    // serverCron
函数每执行一次，这个属性的值就增一。
    int cronloops;
    // ...
};
```

cronloops属性目前在服务器中的唯一作用，就是在复制模块中实现“每执行serverCron函数N次就执行一次指定代码”的功能，方法如以下伪代码所示：

```
if cronloops % N == 0:
    #
执行指定代码...
```

14.3 初始化服务器

一个Redis服务器从启动到能够接受客户端的命令请求，需要经过一系列的初始化和设置过程，比如初始化服务器状态，接受用户指定的服务器配置，创建相应的数据结构和网络连接等等，本节接下来的内容将对服务器的整个初始化过程进行详细的介绍。

14.3.1 初始化服务器状态结构

初始化服务器的第一步就是创建一个struct redisServer类型的实例变量server作为服务器的状态，并为结构中的各个属性设置默认值。

初始化server变量的工作由redis.c/initServerConfig函数完成，以下是这个函数最开头的一部分代码：

```
void initServerConfig(void){
    //
设置服务器的运行id
    getRandomHexChars(server.runid,REDIS_RUN_ID_SIZE);
    //
为运行id
加上结尾字符
    server.runid[REDIS_RUN_ID_SIZE] = '\0';
    //
设置默认配置文件路径
    server.configfile = NULL;
    //
设置默认服务器频率
    server.hz = REDIS_DEFAULT_HZ;
    //
设置服务器的运行架构
    server.arch_bits = (sizeof(long) == 8) ? 64 : 32;
    //
设置默认服务器端口号
    server.port = REDIS_SERVERPORT;
    // ...
}
```

以下是initServerConfig函数完成的主要工作：

·设置服务器的运行ID。

·设置服务器的默认运行频率。

·设置服务器的默认配置文件路径。

·设置服务器的运行架构。

·设置服务器的默认端口号。

·设置服务器的默认RDB持久化条件和AOF持久化条件。

·初始化服务器的LRU时钟。

·创建命令表。

initServerConfig函数设置的服务器状态属性基本都是一些整数、浮点数、或者字符串属性，除了命令表之外，initServerConfig函数没有创建服务器状态的其他数据结构，数据库、慢查询日志、Lua环境、共享对象这些数据结构在之后的步骤才会被创建出来。

当initServerConfig函数执行完毕之后，服务器就可以进入初始化的第二个阶段——载入配置选项。

14.3.2 载入配置选项

在启动服务器时，用户可以通过给定配置参数或者指定配置文件来修改服务器的默认配置。举个例子，如果我们在终端中输入：

```
$ redis-server --port 10086
```

那么我们就通过给定配置参数的方式，修改了服务器的运行端口号。另外，如果我们在终端中输入：

```
$ redis-server redis.conf
```

并且redis.conf文件中包含以下内容：

```
#
将服务器的数据库数量设置为 32
个
databases 32
#
关闭 RDB
文件的压缩功能
rdbcompression no
```

那么我们就通过指定配置文件的方式修改了服务器的数据库数量，以及RDB持久化模块的压缩功能。

服务器在用initServerConfig函数初始化完server变量之后，就会开始载入用户给定的配置参数和配置文件，并根据用户设定的配置，对server变量相关属性的值进行修改。

例如，在初始化server变量时，程序会为决定服务器端口号的port属性设置默认值：

```
void initServerConfig(void){
    // ...
    //
默认值为6379
    server.port = REDIS_SERVERPORT;
    // ...
}
```

不过，如果用户在启动服务器时为配置选项port指定了新值10086，那么server.port属性的值就会被更新为10086，这将使得服务器的端口号从默认的6379变为用户指定的10086。

例如，在初始化server变量时，程序会为决定数据库数量的dbnum属性设置默认值：

```
void initServerConfig(void){
    // ...
    //
默认值为16
    server.dbnum = REDIS_DEFAULT_DBNUM;
    // ...
}
```

不过，如果用户在启动服务器时为选项databases设置了值32，那么server.dbnum属性的值就会被更新为32，这将使得服务器的数据库数量从默认的16个变为用户指定的32个。

其他配置选项相关的服务器状态属性的情况与上面列举的port属性和dbnum属性一样：

·如果用户为这些属性的相应选项指定了新的值，那么服务器就使用用户指定的值来更新相应的属性。

·如果用户没有为属性的相应选项设置新的值，那么服务器就沿用之前initServerConfig函数为属性设置的默认值。

服务器在载入用户指定的配置选项，并对server状态进行更新之后，服务器就可以进入初始化的第三个阶段——初始化服务器数据结构。

14.3.3　初始化服务器数据结构

在之前执行initServerConfig函数初始化server状态时，程序只创建了命令表一个数据结构，不过除了命令表之外，服务器状态还包含其他数据结构，比如：

·server.clients链表，这个链表记录了所有与服务器相连的客户端的状态结构，链表的每个节点都包含了一个redisClient结构实例。

·server.db数组，数组中包含了服务器的所有数据库。

·用于保存频道订阅信息的server.pubsub_channels字典，以及用于保存模式订阅信息的server.pubsub_patterns链表。

·用于执行Lua脚本的Lua环境server.lua。

·用于保存慢查询日志的server.slowlog属性。

当初始化服务器进行到这一步，服务器将调用initServer函数，为以上提到的数据结构分配内存，并在有需要时，为这些数据结构设置或者关联初始化值。

服务器到现在才初始化数据结构的原因在于，服务器必须先载入用户指定的配置选项，然后才能正确地对数据结构进行初始化。如果在执行initServerConfig函数时就对数据结构进行初始化，那么一旦用户通过配置选项修改了和数据结构有关的服务器状态属性，服务器就要重新调整和修改已创建的数据结构。为了避免出现这种麻烦的情况，服务器选择了将server状态的初始化分为两步进行，initServerConfig函数主要负责初始化一般属性，而initServer函数主要负责初始化数据结构。

除了初始化数据结构之外，initServer还进行了一些非常重要的设置操作，其中包括：

·为服务器设置进程信号处理器。

·创建共享对象：这些对象包含Redis服务器经常用到的一些值，比如包含"OK"回复的字符串对象，包含"ERR"回复的字符串对象，包含整数1到10000的字符串对象等等，服务器通过重用这些共享对象来避免反复创建相同的对象。

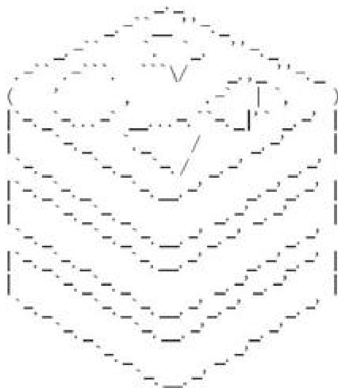
·打开服务器的监听端口，并为监听套接字关联连接应答事件处理器，等待服务器正式运行时接受客户端的连接。

·为serverCron函数创建时间事件，等待服务器正式运行时执行serverCron函数。

·如果AOF持久化功能已经打开，那么打开现有的AOF文件，如果AOF文件不存在，那么创建并打开一个新的AOF文件，为AOF写入做好准备。

·初始化服务器的后台I/O模块（bio），为将来的I/O操作做好准备。

当initServer函数执行完毕之后，服务器将用ASCII字符在日志中打印出Redis的图标，以及Redis的版本号信息：



```
Redis 2.9.11 (b139a2ac/0) 64 bit
Running in stand alone mode
Port: 6379
PID: 5244

http://redis.io

[5244] 21 Nov 22:43:49.084 # Server started, Redis version 2.9.11
```

14.3.4　还原数据库状态

在完成了对服务器状态server变量的初始化之后，服务器需要载入RDB文件或者AOF文件，并根据文件记录的内容来还原服务器的数据库状态。

根据服务器是否启用了AOF持久化功能，服务器载入数据时所使用的目标文件会有所不同：

·如果服务器启用了AOF持久化功能，那么服务器使用AOF文件来还原数据库状态。

·相反地，如果服务器没有启用AOF持久化功能，那么服务器使用RDB文件来还原数据库状态。

当服务器完成数据库状态还原工作之后，服务器将在日志中打印出载入文件并还原数据库状态所耗费的时长：

```
[5244] 21 Nov 22:43:49.084 * DB loaded from disk: 0.068 seconds
```

14.3.5　执行事件循环

在初始化的最后一步，服务器将打印出以下日志：

```
[5244] 21 Nov 22:43:49.084 * The server is now ready to accept connections on port 63
```

并开始执行服务器的事件循环（loop）。

至此，服务器的初始化工作圆满完成，服务器现在开始可以接受客户端的连接请求，并处理客户端发来的命令请求了。

14.4　重点回顾

·一个命令请求从发送到完成主要包括以下步骤：1）客户端将命令请求发送给服务器；2）服务器读取命令请求，并分析出命令参数；3）命令执行器根据参数查找命令的实现函数，然后执行实现函数并得出命令回复；4）服务器将命令回复返回给客户端。

·serverCron函数默认每隔100毫秒执行一次，它的工作主要包括更新服务器状态信息，处理服务器接收的SIGTERM信号，管理客户端资源和数据库状态，检查并执行持久化操作等等。

·服务器从启动到能够处理客户端的命令请求需要执行以下步骤：1）初始化服务器状态；2）载入服务器配置；3）初始化服务器数据结构；4）还原数据库状态；5）执行事件循环。

第三部分　多机数据库的实现

第15章　复制

第16章　Sentinel

第17章　集群

第15章　复制

在Redis中，用户可以通过执行SLAVEOF命令或者设置slaveof选项，让一个服务器去复制（replicate）另一个服务器，我们称呼被复制的服务器为主服务器（master），而对主服务器进行复制的服务器则被称为从服务器（slave），如图15-1所示。



图15-1　主服务器和从服务器

假设现在有两个Redis服务器，地址分别为127.0.0.1:6379和127.0.0.1:12345，如果我们向服务器127.0.0.1:12345发送以下命令：

```
127.0.0.1:12345> SLAVEOF 127.0.0.1 6379
OK
```

那么服务器127.0.0.1:12345将成为127.0.0.1:6379的从服务器，而服务器127.0.0.1:6379则会成为127.0.0.1:12345的主服务器。

进行复制中的主从服务器双方的数据库将保存相同的数据，概念上将这种现象称作“数据库状态一致”，或者简称“一致”。

比如说，如果我们在主服务器上执行以下命令：

```
127.0.0.1:6379> SET msg "hello world"
OK
```

那么我们应该既可以在主服务器上获取msg键的值：

```
127.0.0.1:6379> GET msg
"hello world"
```

又可以在从服务器上获取msg键的值：

```
127.0.0.1:12345> GET msg
"hello world"
```

另一方面，如果我们在主服务器中删除了键msg：

```
127.0.0.1:6379> DEL msg
(integer) 1
```

那么不仅主服务器上的msg键会被删除：

```
127.0.0.1:6379> EXISTS msg
(integer) 0
```

从服务器上的msg键也应该会被删除：

```
127.0.0.1:12345> EXISTS msg
(integer) 0
```

关于复制的特性和用法还有很多，Redis官方网站上的《复制》文档（http://redis.io/topics/replication）已经做了很详细的介绍，这里不再赘述。

本章首先介绍Redis在2.8版本以前使用的旧版复制功能的实现原理，并说明旧版复制功能在处理断线后重新连接的从服务器时，会遇上怎样的低效情况。

接着，本章将介绍Redis从2.8版本开始使用的新版复制功能是如何通过部分重同步来解决旧版复制功能的低效问题的，并说明部分重同步的实现原理。

在此之后，本章将列举SLAVEOF命令的具体实现步骤，并在本章最后，说明主从服务器心跳检测机制的实现原理，并对基于心跳检测实现的几个功能进行介绍。

15.1 旧版复制功能的实现

Redis的复制功能分为同步（sync）和命令传播（command propagate）两个操作：

·同步操作用于将从服务器的数据库状态更新至主服务器当前所处的数据库状态。

·命令传播操作则用于在主服务器的数据库状态被修改，导致主从服务器的数据库状态出现不一致时，让主从服务器的数据库重新回到一致状态。

本节接下来将对同步和命令传播两个操作进行详细的介绍。

15.1.1 同步

当客户端向从服务器发送SLAVEOF命令，要求从服务器复制主服务器时，从服务器首先需要执行同步操作，也即是，将从服务器的数据库状态更新至主服务器当前所处的数据库状态。

从服务器对主服务器的同步操作需要通过向主服务器发送SYNC命令来完成，以下是SYNC命令的执行步骤：

1）从服务器向主服务器发送SYNC命令。

2）收到SYNC命令的主服务器执行BGSAVE命令，在后台生成一个RDB文件，并使用一个缓冲区记录从现在开始执行的所有写命令。

3）当主服务器的BGSAVE命令执行完毕时，主服务器会将BGSAVE命令生成的RDB文件发送给从服务器，从服务器接收并载入这个RDB文件，将自己的数据库状态更新至主服务器执行BGSAVE命令时的数据库状态。

4）主服务器将记录在缓冲区里面的所有写命令发送给从服务器，从服务器执行这些写命令，将自己的数据库状态更新至主服务器数据库当前所处的状态。

图15-2展示了SYNC命令执行期间，主从服务器的通信过程。



图15-2　主从服务器在执行SYNC命令期间的通信过程

表15-1展示了一个主从服务器进行同步的例子。

表15-1　主从服务器的同步过程

时间	主服务器	从服务器
T0	服务器启动	服务器启动
T1	执行 SET k1 v1	
T2	执行 SET k2 v2	
T3	执行 SET k3 v3	
T4		向主服务器发送 *SYNC* 命令
T5	接收到从服务器发来的 *SYNC* 命令，执行 *BGSAVE* 命令，创建包含键 k1、k2、k3 的 RDB 文件，并使用缓冲区记录接下来执行的所有写命令	
T6	执行 SET k4 v4，并将这个命令记录到缓冲区里面	
T7	执行 SET k5 v5，并将这个命令记录到缓冲区里面	
T8	*BGSAVE* 命令执行完毕，向从服务器发送 RDB 文件	
T9		接收并载入主服务器发来的 RDB 文件，获得 k1、k2、k3 三个键
T10	向从服务器发送缓冲区中保存的写命令 SET k4 v4 和 SET k5 v5	
T11		接收并执行主服务器发来的两个 *SET* 命令，得到 k4 和 k5 两个键
T12	同步完成，现在主从服务器两者的数据库都包含了键 k1、k2、k3、k4 和 k5	同步完成，现在主从服务器两者的数据库都包含了键 k1、k2、k3、k4 和 k5

15.1.2　命令传播

在同步操作执行完毕之后，主从服务器两者的数据库将达到一致状态，但这种一致并不是一成不变的，每当主服务器执行客户端发送的写命令时，主服务器的数据库就有可能会被修改，并导致主从服务器状态

不再一致。

举个例子，假设一个主服务器和一个从服务器刚刚完成同步操作，它们的数据库都保存了相同的五个键k1至k5，如图15-3所示。

如果这时，客户端向主服务器发送命令DEL k3，那么主服务器在执行完这个DEL命令之后，主从服务器的数据库将出现不一致：主服务器的数据库已经不再包含键k3，但这个键却仍然包含在从服务器的数据库里面，如图15-4所示。

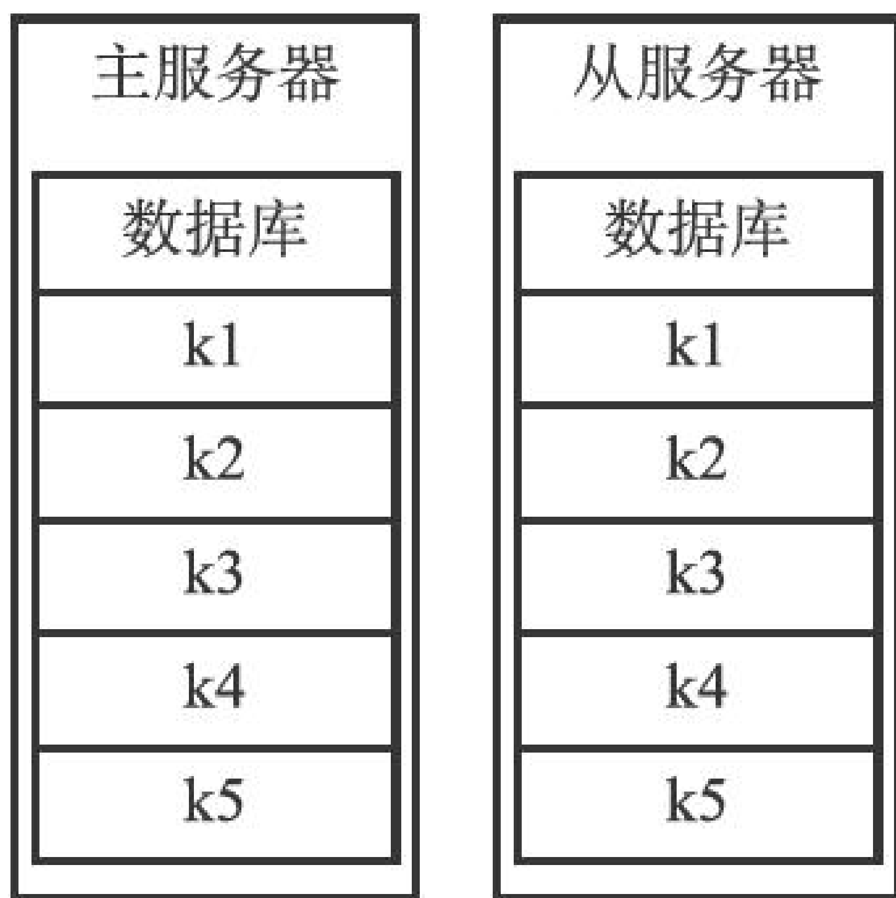


图15-3　处于一致状态的主从服务器

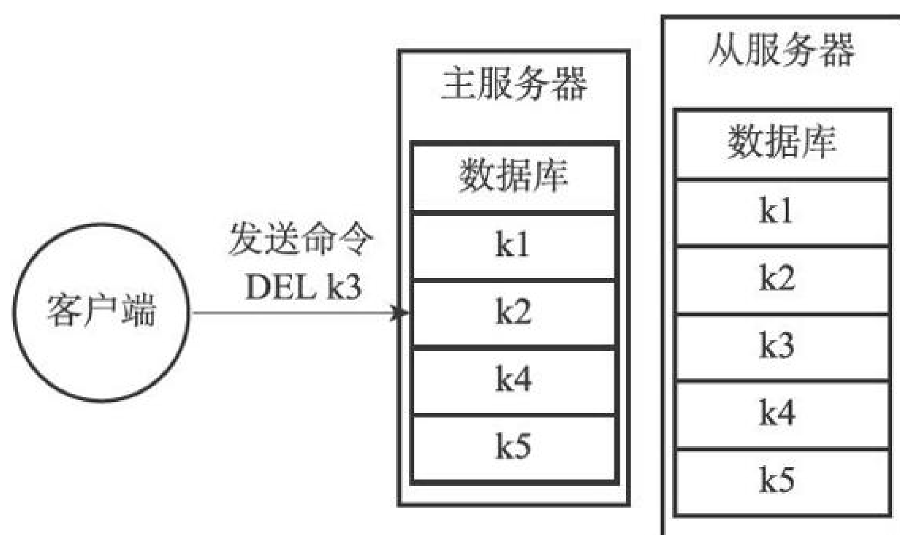


图15-4　处于不一致状态的主从服务器

为了让主从服务器再次回到一致状态，主服务器需要对从服务器执行命令传播操作：主服务器会将自己执行的写命令，也即是造成主从服务器不一致的那条写命令，发送给从服务器执行，当从服务器执行了相同的写命令之后，主从服务器将再次回到一致状态。

在上面的例子中，主服务器因为执行了命令DEL k3而导致主从服务器不一致，所以主服务器将向从服务器发送相同的命令DEL k3。当从服务器执行完这个命令之后，主从服务器将再次回到一致状态，现在主从服务器两者的数据库都不再包含键k3了，如图15-5所示。

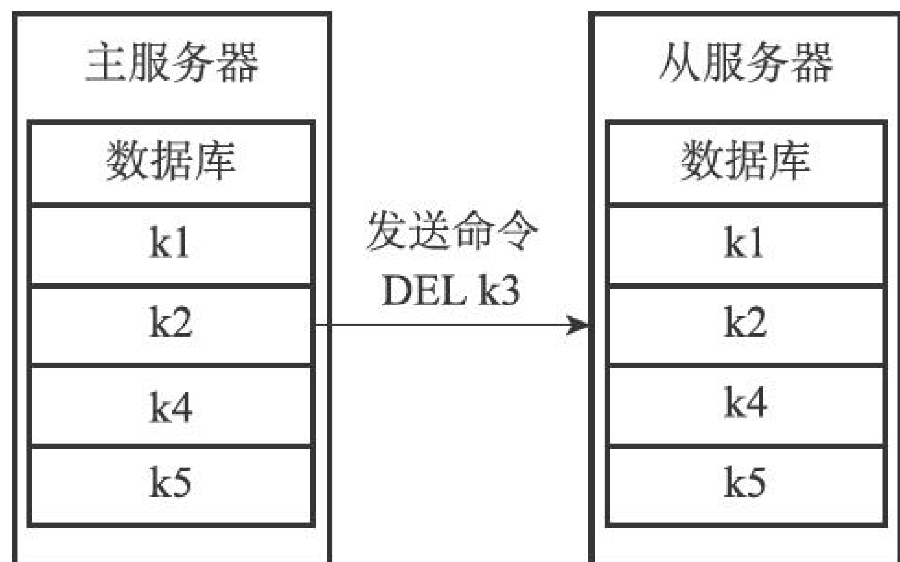


图15-5　主服务器向从服务器发送命令

15.2 旧版复制功能的缺陷

在Redis中，从服务器对主服务器的复制可以分为以下两种情况：

·初次复制：从服务器以前没有复制过任何主服务器，或者从服务器当前要复制的主服务器和上一次复制的主服务器不同。

·断线后重复制：处于命令传播阶段的主从服务器因为网络原因而中断了复制，但从服务器通过自动重连接重新连上了主服务器，并继续复制主服务器。

对于初次复制来说，旧版复制功能能够很好地完成任务，但对于断线后重复制来说，旧版复制功能虽然也能让主从服务器重新回到一致状态，但效率却非常低。

要理解这一情况，请看表15-2展示的断线后重复制例子。

表15-2　从服务器在断线之后重新复制主服务器的例子

时间	主服务器	从服务器
T0	主从服务器完成同步	主从服务器完成同步
T1	执行并传播 SET k1 v1	执行主服务器传来的 SET k1 v1
T2	执行并传播 SET k2 v2	执行主服务器传来的 SET k2 v2
…	…	…
T10085	执行并传播 SET k10085 v10085	执行主服务器传来的 SET k10085 v10085
T10086	执行并传播 SET k10086 v10086	执行主服务器传来的 SET k10086 v10086
T10087	主从服务器连接断开	主从服务器连接断开
T10088	执行 SET k10087 v10087	断线中，尝试重新连接主服务器
T10089	执行 SET k10088 v10088	断线中，尝试重新连接主服务器
T10090	执行 SET k10089 v10089	断线中，尝试重新连接主服务器
T10091	主从服务器重新连接	主从服务器重新连接
T10092		向主服务器发送 *SYNC* 命令
T10093	接收到从服务器发来的 *SYNC* 命令，执行 *BGSAVE* 命令，创建包含键 k1 至键 k10089 的 RDB 文件，并使用缓冲区记录接下来执行的所有写命令	
T10094	*BGSAVE* 命令执行完毕，向从服务器发送 RDB 文件	
T10095		接收并载入主服务器发来的 RDB 文件，获得键 k1 至键 k10089
T10096	因为在 *BGSAVE* 命令执行期间，主服务器没有执行任何写命令，所以跳过发送缓冲区包含的写命令这一步	
T10097	主从服务器再次完成同步	主从服务器再次完成同步

在时间T10091，从服务器终于重新连接上主服务器，因为这时主从

服务器的状态已经不再一致，所以从服务器将向主服务器发送SYNC命令，而主服务器会将包含键k1至键k10089的RDB文件发送给从服务器，从服务器通过接收和载入这个RDB文件来将自己的数据库更新至主服务器数据库当前所处的状态。

虽然再次发送SYNC命令可以让主从服务器重新回到一致状态，但如果我们仔细研究这个断线重复制过程，就会发现传送RDB文件这一步实际上并不是非做不可的：

·主从服务器在时间T0至时间T10086中一直处于一致状态，这两个服务器保存的数据大部分都是相同的。

·从服务器想要将自己更新至主服务器当前所处的状态，真正需要的是主从服务器连接中断期间，主服务器新添加的k10087、k10088、k10089三个键的数据。

·可惜的是，旧版复制功能并没有利用以上列举的两点条件，而是继续让主服务器生成并向从服务器发送包含键k1至键k10089的RDB文件，但实际上RDB文件包含的键k1至键k10086的数据对于从服务器来说都是不必要的。

上面给出的例子可能有一点理想化，因为在主从服务器断线期间，主服务器执行的写命令可能会有成百上千个之多，而不仅仅是两三个写命令。但总的来说，主从服务器断开的时间越短，主服务器在断线期间执行的写命令就越少，而执行少量写命令所产生的数据量通常比整个数据库的数据量要少得多，在这种情况下，为了让从服务器补足一小部分缺失的数据，却要让主从服务器重新执行一次SYNC命令，这种做法无疑是非常低效的。

SYNC命令是一个非常耗费资源的操作

每次执行SYNC命令，主从服务器需要执行以下动作：

1）主服务器需要执行BGSAVE命令来生成RDB文件，这个生成操作会耗费主服务器大量的CPU、内存和磁盘I/O资源。

2）主服务器需要将自己生成的RDB文件发送给从服务器，这个发送操作会耗费主从服务器大量的网络资源（带宽和流量），并对主服务器响应命令请求的时间产生影响。

3）接收到RDB文件的从服务器需要载入主服务器发来的RDB文件，并且在载入期间，从服务器会因为阻塞而没办法处理命令请求。

因为SYNC命令是一个如此耗费资源的操作，所以Redis有必要保证在真正有需要时才执行SYNC命令。

15.3 新版复制功能的实现

为了解决旧版复制功能在处理断线重复制情况时的低效问题，Redis从2.8版本开始，使用PSYNC命令代替SYNC命令来执行复制时的同步操作。

PSYNC命令具有完整重同步（full resynchronization）和部分重同步（partial resynchronization）两种模式：

·其中完整重同步用于处理初次复制情况：完整重同步的执行步骤和SYNC命令的执行步骤基本一样，它们都是通过让主服务器创建并发送RDB文件，以及向从服务器发送保存在缓冲区里面的写命令来进行同步。

·而部分重同步则用于处理断线后重复制情况：当从服务器在断线后重新连接主服务器时，如果条件允许，主服务器可以将主从服务器连接断开期间执行的写命令发送给从服务器，从服务器只要接收并执行这些写命令，就可以将数据库更新至主服务器当前所处的状态。

PSYNC命令的部分重同步模式解决了旧版复制功能在处理断线后重复制时出现的低效情况，表15-3展示了如何使用PSYNC命令高效地处理上一节展示的断线后复制情况。

表15-3 使用PSYNC命令来进行断线后重复制

时间	主服务器	从服务器
T0	主从服务器完成同步	主从服务器完成同步
T1	执行并传播 SET k1 v1	执行主服务器传来的 SET k1 v1
T2	执行并传播 SET k2 v2	执行主服务器传来的 SET k2 v2
...	...	...
T10085	执行并传播 SET k10085 v10085	执行主服务器传来的SET k10085 v10085
T10086	执行并传播 SET k10086 v10086	执行主服务器传来的SET k10086 v10086
T10087	主从服务器连接断开	主从服务器连接断开
T10088	执行 SET k10087 v10087	断线中，尝试重新连接主服务器
T10089	执行 SET k10088 v10088	断线中，尝试重新连接主服务器
T10090	执行 SET k10089 v10089	断线中，尝试重新连接主服务器
T10091	主从服务器重新连接	主从服务器重新连接
T10092		向主服务器发送 PSYNC 命令

T10093	向从服务器返回 +CONTINUE 回复，表示执行部分重同步	
T10094		接收 +CONTINUE 回复，准备执行部分重同步
T10095	向从服务器发送 SET k10087 v10087、SET k10088 v10088、SET k10089 v10089 三个命令	
T10096		接收并执行主服务器传来的三个 *SET* 命令
T10097	主从服务器再次完成同步	主从服务器再次完成同步

对比一下SYNC命令和PSYNC命令处理断线重复制的方法，不难看出，虽然SYNC命令和PSYNC命令都可以让断线的主从服务器重新回到一致状态，但执行部分重同步所需的资源比起执行SYNC命令所需的资源要少得多，完成同步的速度也快得多。执行SYNC命令需要生成、传送和载入整个RDB文件，而部分重同步只需要将从服务器缺少的写命令发送给从服务器执行就可以了。

图15-6展示了主从服务器在执行部分重同步时的通信过程。

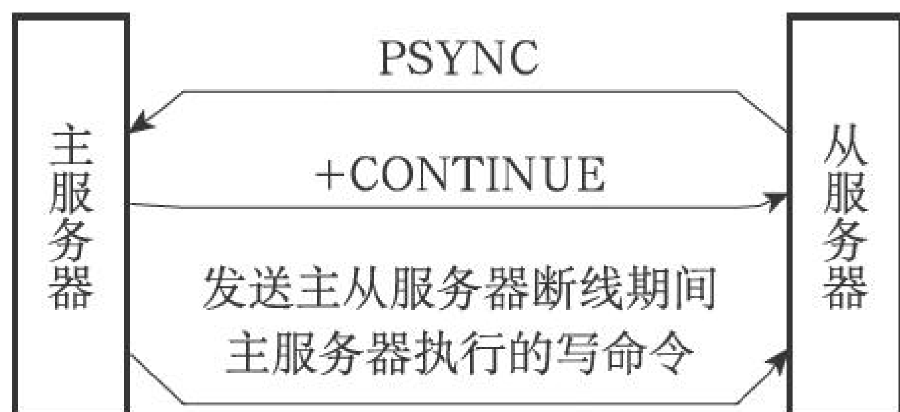


图15-6　主从服务器执行部分重同步的过程

15.4　部分重同步的实现

在了解了PSYNC命令的由来，以及部分重同步的工作方式之后，是时候来介绍一下部分重同步的实现细节了。

部分重同步功能由以下三个部分构成：

·主服务器的复制偏移量（replication　offset）和从服务器的复制偏移量。

·主服务器的复制积压缓冲区（replication backlog）。

·服务器的运行ID（run ID）。

以下三个小节将分别介绍这三个部分。

15.4.1　复制偏移量

执行复制的双方——主服务器和从服务器会分别维护一个复制偏移量：

·主服务器每次向从服务器传播N个字节的数据时，就将自己的复制偏移量的值加上N。

·从服务器每次收到主服务器传播来的N个字节的数据时，就将自己的复制偏移量的值加上N。

在图15-7所示的例子中，主从服务器的复制偏移量的值都为10086。

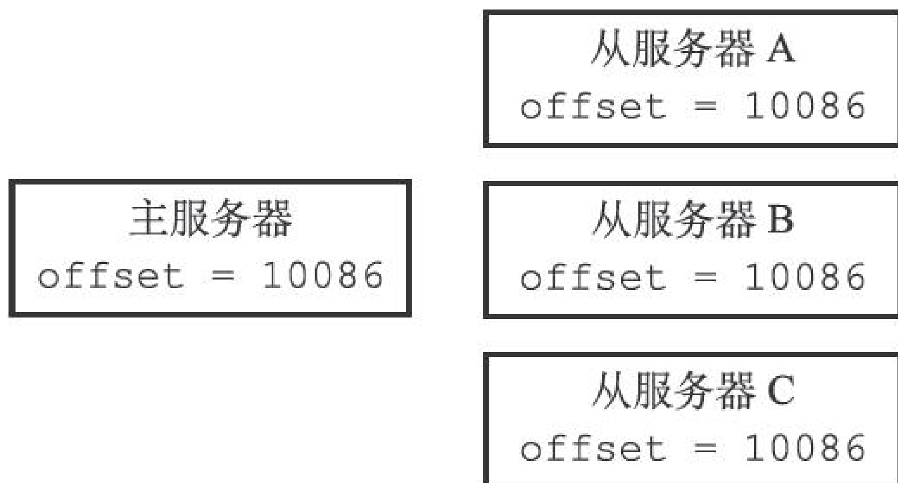


图15-7　拥有相同偏移量的主服务器和它的三个从服务器

如果这时主服务器向三个从服务器传播长度为33字节的数据，那么主服务器的复制偏移量将更新为10086＋33＝10119，而三个从服务器在接收到主服务器传播的数据之后，也会将复制偏移量更新为10119，如图15-8所示。

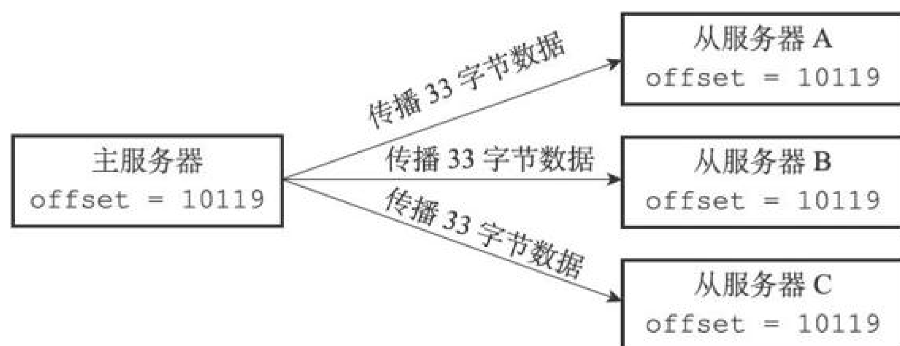


图15-8　更新偏移量之后的主从服务器

通过对比主从服务器的复制偏移量，程序可以很容易地知道主从服务器是否处于一致状态：

·如果主从服务器处于一致状态，那么主从服务器两者的偏移量总是相同的。

·相反，如果主从服务器两者的偏移量并不相同，那么说明主从服务器并未处于一致状态。

考虑以下这个例子：假设如图15-7所示，主从服务器当前的复制偏移量都为10086，但是就在主服务器要向从服务器传播长度为33字节的数据之前，从服务器A断线了，那么主服务器传播的数据将只有从服务器B和从服务器C能收到，在这之后，主服务器、从服务器B和从服务器C三个服务器的复制偏移量都将更新为10119，而断线的从服务器A的复制偏移量仍然停留在10086，这说明从服务器A与主服务器并不一致，如图15-9所示。

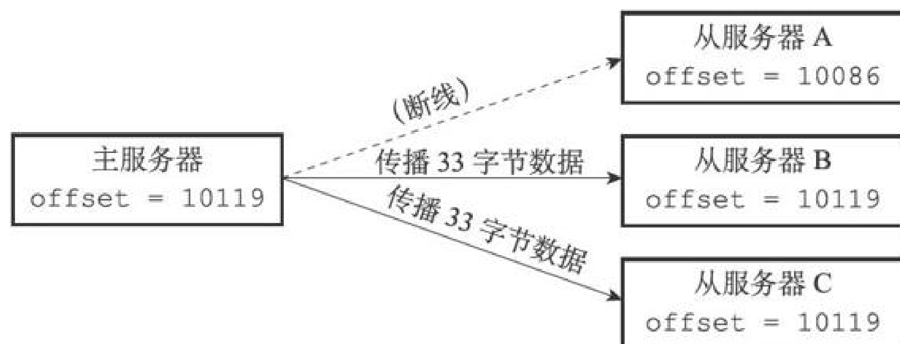


图15-9　因为断线而处于不一致状态的从服务器A

假设从服务器A在断线之后就立即重新连接主服务器，并且成功，那么接下来，从服务器将向主服务器发送PSYNC命令，报告从服务器A当前的复制偏移量为10086，那么这时，主服务器应该对从服务器执行完整重同步还是部分重同步呢？如果执行部分重同步的话，主服务器又如何补偿从服务器A在断线期间丢失的那部分数据呢？以上问题的答案都和复制积压缓冲区有关。

15.4.2　复制积压缓冲区

复制积压缓冲区是由主服务器维护的一个固定长度（fixed-size）先进先出（FIFO）队列，默认大小为1MB。

固定长度先进先出队列

固定长度先进先出队列的入队和出队规则跟普通的先进先出队列一样：新元素从一边进入队列，而旧元素从另一边弹出队列。

和普通先进先出队列随着元素的增加和减少而动态调整长度不同，固定长度先进先出队列的长度是固定的，当入队元素的数量大于队列长度时，最先入队的元素会被弹出，而新元素会被放入队列。

举个例子，如果我们要将'h'、'e'、'l'、'l'、'o'五个字符放进一个长度为3的固定长度先进先出队列里面，那么'h'、'e'、'l'三个字符将首先被放入队列：

['h'，'e'，'l']

但是当后一个'l'字符要进入队列时，队首的'h'字符将被弹出，队列变成：

['e'，'l'，'l']

接着，'o'的入队会引起'e'的出队，队列变成：

['l'，'l'，'o']

以上就是固定长度先进先出队列的运作方式。

当主服务器进行命令传播时，它不仅会将写命令发送给所有从服务器，还会将写命令入队到复制积压缓冲区里面，如图15-10所示。

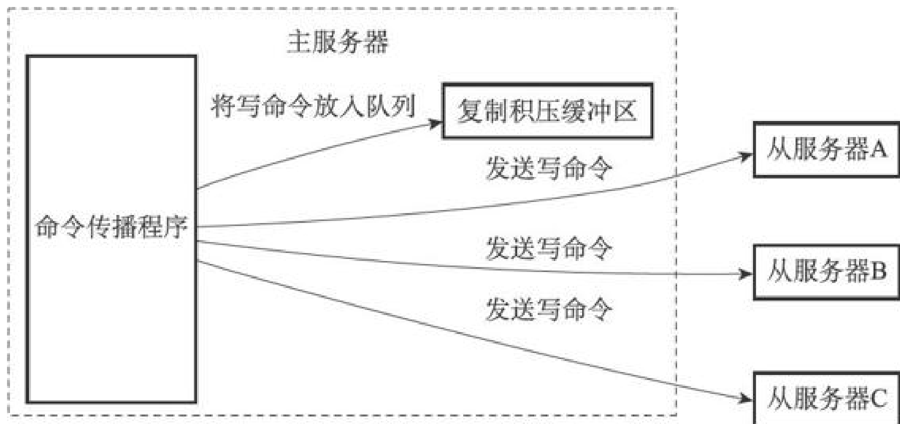


图15-10　主服务器向复制积压缓冲区和所有从服务器传播写命令数据

因此，主服务器的复制积压缓冲区里面会保存着一部分最近传播的写命令，并且复制积压缓冲区会为队列中的每个字节记录相应的复制偏移量，就像表15-4展示的那样。

表15-4　复制积压缓冲区的构造

偏移量	…	10087	10088	10089	10090	10091	10092	10093	10094	10095	10096	10097	…
字节值	…	'*'	3	'\r'	'\n'	'$'	3	'\r'	'\n'	'S'	'E'	'T'	…

当从服务器重新连上主服务器时，从服务器会通过PSYNC命令将自己的复制偏移量offset发送给主服务器，主服务器会根据这个复制偏移量来决定对从服务器执行何种同步操作：

·如果offset偏移量之后的数据（也即是偏移量offset＋1开始的数据）仍然存在于复制积压缓冲区里面，那么主服务器将对从服务器执行部分重同步操作。

·相反，如果offset偏移量之后的数据已经不存在于复制积压缓冲区，那么主服务器将对从服务器执行完整重同步操作。

回到之前图15-9展示的断线后重连接例子：

·当从服务器A断线之后，它立即重新连接主服务器，并向主服务器发送PSYNC命令，报告自己的复制偏移量为10086。

·主服务器收到从服务器发来的PSYNC命令以及偏移量10086之后，主服务器将检查偏移量10086之后的数据是否存在于复制积压缓冲区里面，结果发现这些数据仍然存在，于是主服务器向从服务器发送＋CONTINUE回复，表示数据同步将以部分重同步模式来进行。

·接着主服务器会将复制积压缓冲区10086偏移量之后的所有数据（偏移量为10087至10119）都发送给从服务器。

·从服务器只要接收这33字节的缺失数据，就可以回到与主服务器一致的状态，如图15-11所示。

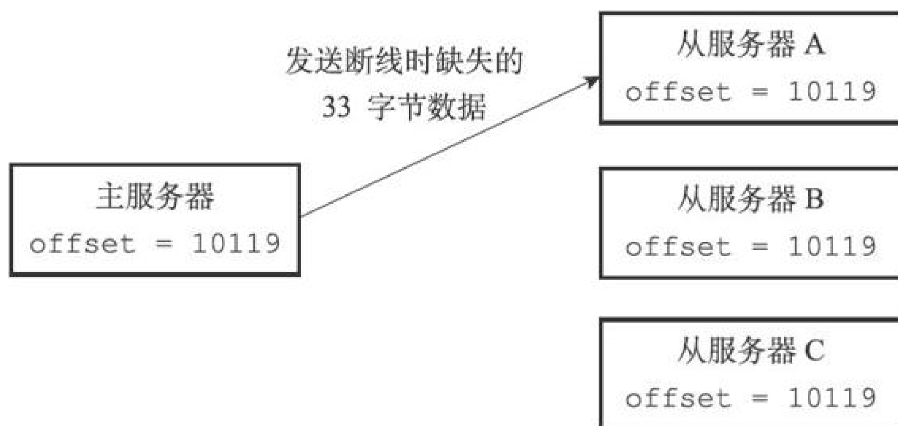


图15-11　主服务器向从服务器发送缺失的数据

根据需要调整复制积压缓冲区的大小

Redis为复制积压缓冲区设置的默认大小为1MB，如果主服务器需要执行大量写命令，又或者主从服务器断线后重连接所需的时间比较长，那么这个大小也许并不合适。如果复制积压缓冲区的大小设置得不恰当，那么PSYNC命令的复制重同步模式就不能正常发挥作用，因此，正确估算和设置复制积压缓冲区的大小非常重要。

复制积压缓冲区的最小大小可以根据公式second*write_size_per_second来估算：

·其中second为从服务器断线后重新连接上主服务器所需的平均时间（以秒计算）。

·而write_size_per_second则是主服务器平均每秒产生的写命令数据量（协议格式的写命令的长度总和）。

例如，如果主服务器平均每秒产生1 MB的写数据，而从服务器断线之后平均要5秒才能重新连接上主服务器，那么复制积压缓冲区的大小就不能低于5MB。

为了安全起见，可以将复制积压缓冲区的大小设为2*second*write_size_per_second，这样可以保证绝大部分断线情况都能用部分重同步来处理。

至于复制积压缓冲区大小的修改方法，可以参考配置文件中关于repl-backlog-size选项的说明。

15.4.3　服务器运行ID

除了复制偏移量和复制积压缓冲区之外，实现部分重同步还需要用到服务器运行ID（run ID）：

·每个Redis服务器，不论主服务器还是从服务，都会有自己的运行ID。

·运行ID在服务器启动时自动生成，由40个随机的十六进制字符组成，例如53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3。

当从服务器对主服务器进行初次复制时，主服务器会将自己的运行ID传送给从服务器，而从服务器则会将这个运行ID保存起来。

当从服务器断线并重新连上一个主服务器时，从服务器将向当前连接的主服务器发送之前保存的运行ID：

·如果从服务器保存的运行ID和当前连接的主服务器的运行ID相同，那么说明从服务器断线之前复制的就是当前连接的这个主服务器，主服务器可以继续尝试执行部分重同步操作。

·相反地，如果从服务器保存的运行ID和当前连接的主服务器的运行ID并不相同，那么说明从服务器断线之前复制的主服务器并不是当前连接的这个主服务器，主服务器将对从服务器执行完整重同步操作。

举个例子，假设从服务器原本正在复制一个运行ID为53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3的主服务器，那么在网络断开，从服务器重新连接上主服务器之后，从服务器将向主服务器发送这个运行ID，主服务器根据自己的运行ID是否53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3来判断是执行部分重同步还是执行完整重同步。

15.5　PSYNC命令的实现

到目前为止，本章在介绍PSYNC命令时一直没有说明PSYNC命令的参数以及返回值，因为那时我们还未了解服务器运行ID、复制偏移量、复制积压缓冲区这些东西，在学习了部分重同步的实现原理之后，我们现在可以来了解PSYNC命令的完整细节了。

PSYNC命令的调用方法有两种：

·如果从服务器以前没有复制过任何主服务器，或者之前执行过SLAVEOF　no　one命令，那么从服务器在开始一次新的复制时将向主服务器发送PSYNC　？　-1命令，主动请求主服务器进行完整重同步（因为这时不可能执行部分重同步）。

·相反地，如果从服务器已经复制过某个主服务器，那么从服务器在开始一次新的复制时将向主服务器发送PSYNC　＜runid＞　＜offset＞命令：其中runid是上一次复制的主服务器的运行ID，而offset则是从服务器当前的复制偏移量，接收到这个命令的主服务器会通过这两个参数来判断应该对从服务器执行哪种同步操作。

根据情况，接收到PSYNC命令的主服务器会向从服务器返回以下三种回复的其中一种：

·如果主服务器返回＋FULLRESYNC　＜runid＞　＜offset＞回复，那么表示主服务器将与从服务器执行完整重同步操作：其中runid是这个主服务器的运行ID，从服务器会将这个ID保存起来，在下一次发送PSYNC命令时使用；而offset则是主服务器当前的复制偏移量，从服务器会将这个值作为自己的初始化偏移量。

·如果主服务器返回＋CONTINUE回复，那么表示主服务器将与从服务器执行部分重同步操作，从服务器只要等着主服务器将自己缺少的那部分数据发送过来就可以了。

·如果主服务器返回-ERR回复，那么表示主服务器的版本低于Redis 2.8，它识别不了PSYNC命令，从服务器将向主服务器发送SYNC命令，并与主服务器执行完整同步操作。

流程图15-12总结了PSYNC命令执行完整重同步和部分重同步时可能遇上的情况。

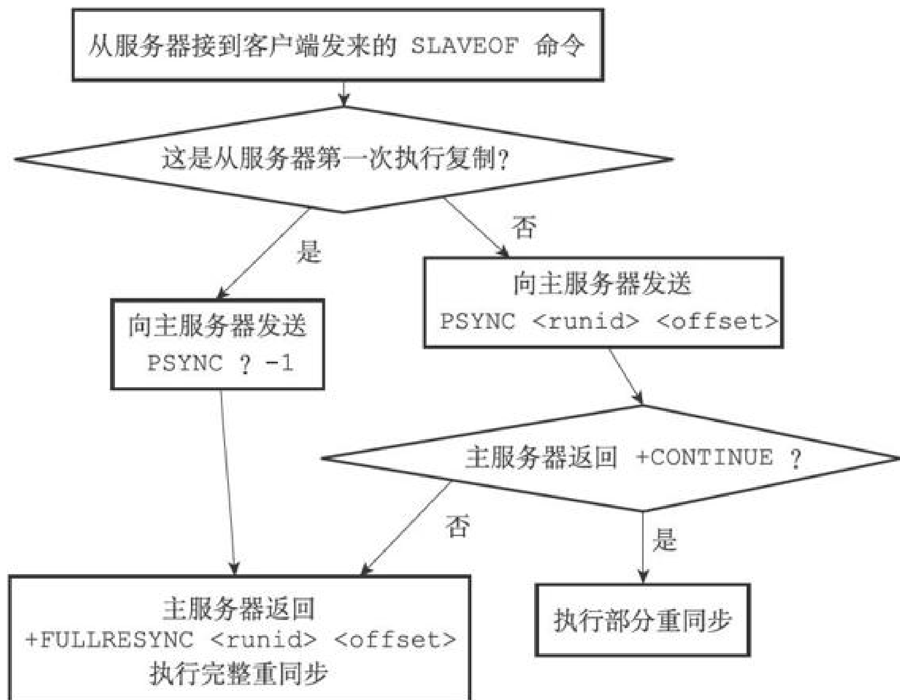


图15-12 PSYNC执行完整重同步和部分重同步时可能遇上的情况

为了熟悉PSYNC命令的用法，让我们来看一个完整的复制——网络中断——重复制例子。

首先，假设有两个Redis服务器，它们的版本都是Redis 2.8，其中主服务器的地址为127.0.0.1:6379，从服务器的地址为127.0.0.1:12345。

如果客户端向从服务器发送命令SLAVEOF 127.0.0.1 6379，并且假设从服务器是第一次执行复制操作，那么从服务器将向主服务器发送PSYNC ? -1命令，请求主服务器执行完整重同步操作。

主服务器在收到完整重同步请求之后，将在后台执行BGSAVE命令，并向从服务器返回＋FULLRESYNC 53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3 10086回复，其中53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3是主服务器的运行ID，而10086则是主服务器当前的复制偏移量。

假设完整重同步成功执行，并且主从服务器在一段时间之后仍然保持一致，但是在复制偏移量为20000的时候，主从服务器之间的网络连接中断了，这时从服务器将重新连接主服务器，并再次对主服务器进行复制。

因为之前曾经对主服务器进行过复制，所以从服务器将向主服务器

发送命令PSYNC 53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3 20000，请求进行部分重同步。

主服务器在接收到从服务器的PSYNC命令之后，首先对比从服务器传来的运行ID53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3和主服务器自身的运行ID，结果显示该ID和主服务器的运行ID相同，于是主服务器继续读取从服务器传来的偏移量20000，检查偏移量为20000之后的数据是否存在于复制积压缓冲区里面，结果发现数据仍然存在。

确认运行ID相同并且数据存在之后，主服务器将向从服务器返回+CONTINUE回复，表示将与从服务器执行部分重同步操作，之后主服务器会将保存在复制积压缓冲区20000偏移量之后的所有数据发送给从服务器，主从服务器将再次回到一致状态。

15.6 复制的实现

通过向从服务器发送SLAVEOF命令，我们可以让一个从服务器去复制一个主服务器：

```
SLAVEOF <master_ip> <master_port>
```

本节将以从服务器127.0.0.1:12345接收到命令：

```
SLAVEOF 127.0.0.1 6379
```

为例，展示Redis2.8或以上版本的复制功能的详细实现步骤。

15.6.1 步骤1：设置主服务器的地址和端口

当客户端向从服务器发送以下命令时：

```
127.0.0.1:12345> SLAVEOF 127.0.0.1 6379
OK
```

从服务器首先要做的就是将客户端给定的主服务器IP地址127.0.0.1以及端口6379保存到服务器状态的masterhost属性和masterport属性里面：

```
struct redisServer {
    // ...
    //
主服务器的地址
    char *masterhost;
    //
主服务器的端口
    int masterport;
    // ...
};
```

图15-13展示了SLAVEOF命令执行之后，从服务器的服务器状态。

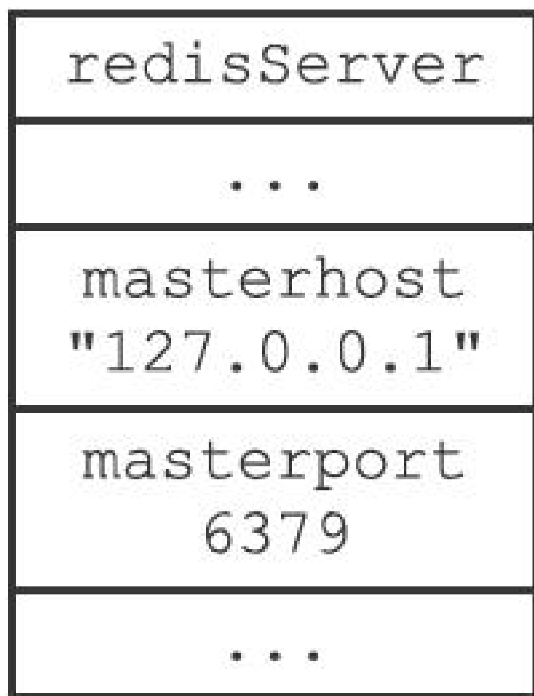


图15-13　从服务器的服务器状态

SLAVEOF命令是一个异步命令，在完成masterhost属性和masterport属性的设置工作之后，从服务器将向发送SLAVEOF命令的客户端返回OK，表示复制指令已经被接收，而实际的复制工作将在OK返回之后才真正开始执行。

15.6.2　步骤2：建立套接字连接

在SLAVEOF命令执行之后，从服务器将根据命令所设置的IP地址和端口，创建连向主服务器的套接字连接，如图15-14所示。



图15-14　从服务器创建连向主服务器的套接字

如果从服务器创建的套接字能成功连接（connect）到主服务器，那么从服务器将为这个套接字关联一个专门用于处理复制工作的文件事件处理器，这个处理器将负责执行后续的复制工作，比如接收RDB文件，以及接收主服务器传播来的写命令，诸如此类。

而主服务器在接受（accept）从服务器的套接字连接之后，将为该

套接字创建相应的客户端状态，并将从服务器看作是一个连接到主服务器的客户端来对待，这时从服务器将同时具有服务器（server）和客户端（client）两个身份：从服务器可以向主服务器发送命令请求，而主服务器则会向从服务器返回命令回复，如图15-15所示。



图15-15　主从服务器之间的关系

因为复制工作接下来的几个步骤都会以从服务器向主服务器发送命令请求的形式来进行，所以理解“从服务器是主服务器的客户端”这一点非常重要。

15.6.3　步骤3：发送PING命令

从服务器成为主服务器的客户端之后，做的第一件事就是向主服务器发送一个PING命令，如图15-16所示。

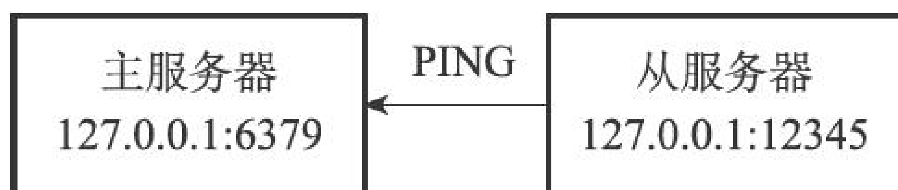


图15-16　从服务器向主服务器发送PING

这个PING命令有两个作用：

·虽然主从服务器成功建立起了套接字连接，但双方并未使用该套接字进行过任何通信，通过发送PING命令可以检查套接字的读写状态是否正常。

·因为复制工作接下来的几个步骤都必须在主服务器可以正常处理命令请求的状态下才能进行，通过发送PING命令可以检查主服务器能否正常处理命令请求。

从服务器在发送PING命令之后将遇到以下三种情况的其中一种：

·如果主服务器向从服务器返回了一个命令回复，但从服务器却不能在规定的时限（timeout）内读取出命令回复的内容，那么表示主从服务器之间的网络连接状态不佳，不能继续执行复制工作的后续步骤。当出现这种情况时，从服务器断开并重新创建连向主服务器的套接字。

·如果主服务器向从服务器返回一个错误，那么表示主服务器暂时没办法处理从服务器的命令请求，不能继续执行复制工作的后续步骤。当出现这种情况时，从服务器断开并重新创建连向主服务器的套接字。比如说，如果主服务器正在处理一个超时运行的脚本，那么当从服务器向主服务器发送PING命令时，从服务器将收到主服务器返回的BUSY Redisis busy running a script.You can only call SCRIPT KILL or SHUTDOWN NOSAVE.错误。

·如果从服务器读取到"PONG"回复，那么表示主从服务器之间的网络连接状态正常，并且主服务器可以正常处理从服务器（客户端）发送的命令请求，在这种情况下，从服务器可以继续执行复制工作的下个步骤。

流程图15-17总结了从服务器在发送PING命令时可能遇到的情况，以及各个情况的处理方式。

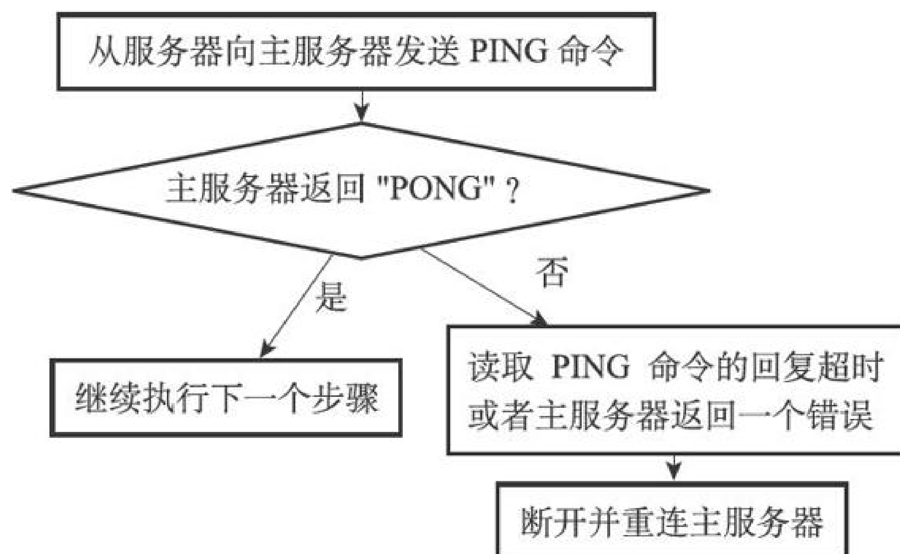


图15-17　从服务器在发送PING命令时可能遇上的情况

15.6.4　步骤4：身份验证

从服务器在收到主服务器返回的"PONG"回复之后，下一步要做的就是决定是否进行身份验证：

·如果从服务器设置了masterauth选项，那么进行身份验证。

·如果从服务器没有设置masterauth选项，那么不进行身份验证。

在需要进行身份验证的情况下，从服务器将向主服务器发送一条AUTH命令，命令的参数为从服务器masterauth选项的值。

举个例子，如果从服务器masterauth选项的值为10086，那么从服务器将向主服务器发送命令AUTH 10086，如图15-18所示。

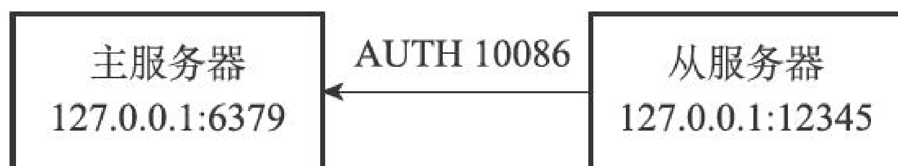


图15-18　从服务器向主服务器验证身份

从服务器在身份验证阶段可能遇到的情况有以下几种：

·如果主服务器没有设置requirepass选项，并且从服务器也没有设置masterauth选项，那么主服务器将继续执行从服务器发送的命令，复制工作可以继续进行。

·如果从服务器通过AUTH命令发送的密码和主服务器requirepass选项所设置的密码相同，那么主服务器将继续执行从服务器发送的命令，复制工作可以继续进行。与此相反，如果主从服务器设置的密码不相同，那么主服务器将返回一个invalid password错误。

·如果主服务器设置了requirepass选项，但从服务器却没有设置masterauth选项，那么主服务器将返回一个NOAUTH错误。另一方面，如果主服务器没有设置requirepass选项，但从服务器却设置了masterauth选项，那么主服务器将返回一个no password is set错误。

所有错误情况都会令从服务器中止目前的复制工作，并从创建套接字开始重新执行复制，直到身份验证通过，或者从服务器放弃执行复制为止。

流程图15-19总结了从服务器在身份验证阶段可能遇到的情况，以及各个情况的处理方式。

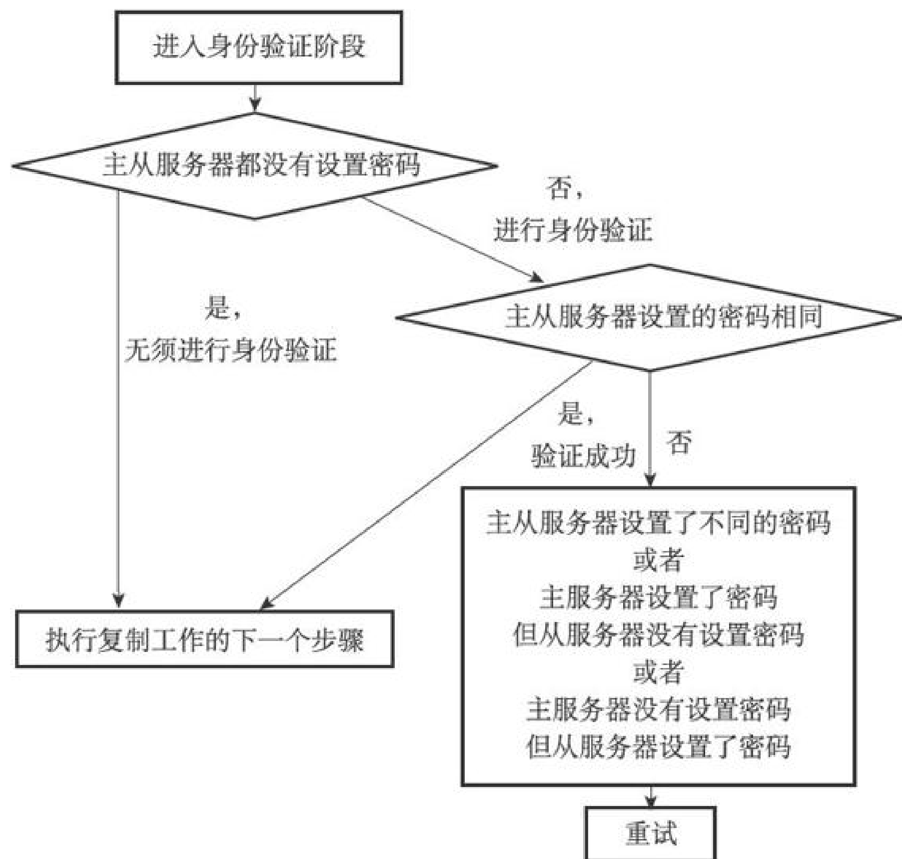


图15-19　从服务器在身份验证阶段可能遇上的情况

15.6.5　步骤5：发送端口信息

在身份验证步骤之后，从服务器将执行命令REPLCONF listening-port ＜port-number＞，向主服务器发送从服务器的监听端口号。

例如在我们的例子中，从服务器的监听端口为12345，那么从服务器将向主服务器发送命令REPLCONF listening-port 12345，如图15-20所示。



图15-20　从服务器向主服务器发送监听端口

主服务器在接收到这个命令之后，会将端口号记录在从服务器所对应的客户端状态的slave_listening_port属性中：

```
typedef struct redisClient {
```

```
    // ...
    //
从服务器的监听端口号
    int slave_listening_port;
    // ...
} redisClient;
```

图15-21展示了客户端状态设置slave_listening_port属性之后的样子。

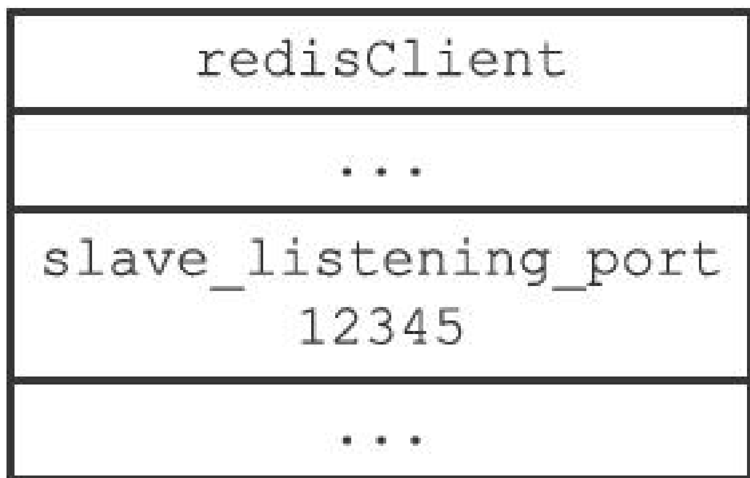


图15-21　用客户端状态记录从服务器的监听端口

slave_listening_port属性目前唯一的作用就是在主服务器执行INFO replication命令时打印出从服务器的端口号。

以下是客户端向例子中的主服务器发送INFO replication命令时得到的回复，其中slave0行的port域显示的就是从服务器所对应客户端状态的slave_listening_port属性的值：

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=12345,status=online,offset=1289,lag=1
master_repl_offset:1289
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:1288
```

15.6.6　步骤6：同步

在这一步，从服务器将向主服务器发送PSYNC命令，执行同步操作，并将自己的数据库更新至主服务器数据库当前所处的状态。

值得一提的是，在同步操作执行之前，只有从服务器是主服务器的客户端，但是在执行同步操作之后，主服务器也会成为从服务器的客户

端：

·如果PSYNC命令执行的是完整重同步操作，那么主服务器需要成为从服务器的客户端，才能将保存在缓冲区里面的写命令发送给从服务器执行。

·如果PSYNC命令执行的是部分重同步操作，那么主服务器需要成为从服务器的客户端，才能向从服务器发送保存在复制积压缓冲区里面的写命令。

因此，在同步操作执行之后，主从服务器双方都是对方的客户端，它们可以互相向对方发送命令请求，或者互相向对方返回命令回复，如图15-22所示。

正因为主服务器成为了从服务器的客户端，所以主服务器才可以通过发送写命令来改变从服务器的数据库状态，不仅同步操作需要用到这一点，这也是主服务器对从服务器执行命令传播操作的基础。



图15-22　主从服务器之间互为客户端

15.6.7　步骤7：命令传播

当完成了同步之后，主从服务器就会进入命令传播阶段，这时主服务器只要一直将自己执行的写命令发送给从服务器，而从服务器只要一直接收并执行主服务器发来的写命令，就可以保证主从服务器一直保持一致了。

以上就是Redis 2.8或以上版本的复制功能的实现步骤。

15.7 心跳检测

在命令传播阶段，从服务器默认会以每秒一次的频率，向主服务器发送命令：

```
REPLCONF ACK <replication_offset>
```

其中replication_offset是从服务器当前的复制偏移量。

发送REPLCONF ACK命令对于主从服务器有三个作用：

·检测主从服务器的网络连接状态。

·辅助实现min-slaves选项。

·检测命令丢失。

以下三个小节将分别介绍这三个作用。

15.7.1 检测主从服务器的网络连接状态

主从服务器可以通过发送和接收REPLCONF ACK命令来检查两者之间的网络连接是否正常：如果主服务器超过一秒钟没有收到从服务器发来的REPLCONF ACK命令，那么主服务器就知道主从服务器之间的连接出现问题了。

通过向主服务器发送INFO replication命令，在列出的从服务器列表的lag一栏中，我们可以看到相应从服务器最后一次向主服务器发送REPLCONF ACK命令距离现在过了多少秒：

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=12345,state=online,offset=211,lag=0   #
刚刚发送过 REPLCONF ACK
命令
slave1:ip=127.0.0.1,port=56789,state=online,offset=197,lag=15   # 15
秒之前发送过REPLCONF ACK
命令
master_repl_offset:211
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:210
```

在一般情况下，lag的值应该在0秒或者1秒之间跳动，如果超过1秒的话，那么说明主从服务器之间的连接出现了故障。

15.7.2 辅助实现min-slaves配置选项

Redis的min-slaves-to-write和min-slaves-max-lag两个选项可以防止主服务器在不安全的情况下执行写命令。

举个例子，如果我们向主服务器提供以下设置：

```
min-slaves-to-write 3
min-slaves-max-lag 10
```

那么在从服务器的数量少于3个，或者三个从服务器的延迟（lag）值都大于或等于10秒时，主服务器将拒绝执行写命令，这里的延迟值就是上面提到的INFO replication命令的lag值。

15.7.3　检测命令丢失

如果因为网络故障，主服务器传播给从服务器的写命令在半路丢失，那么当从服务器向主服务器发送REPLCONF ACK命令时，主服务器将发觉从服务器当前的复制偏移量少于自己的复制偏移量，然后主服务器就会根据从服务器提交的复制偏移量，在复制积压缓冲区里面找到从服务器缺少的数据，并将这些数据重新发送给从服务器。

举个例子，假设有两个处于一致状态的主从服务器，它们的复制偏移量都是200，如图15-23所示。

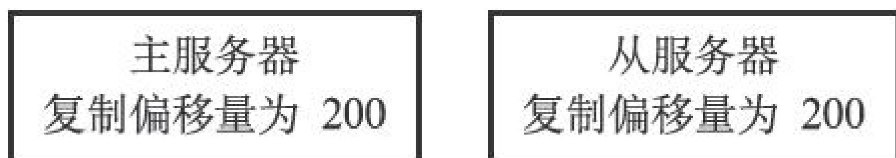


图15-23　主从服务器处于一致状态

如果这时主服务器执行了命令SET key value（协议格式的长度为33字节），将自己的复制偏移量更新到了233，并尝试向从服务器传播命令SET key value，但这条命令却因为网络故障而在传播的途中丢失，那么主从服务器之间的复制偏移量就会出现不一致，主服务器的复制偏移量会被更新为233，而从服务器的复制偏移量仍然为200，如图15-24所示。

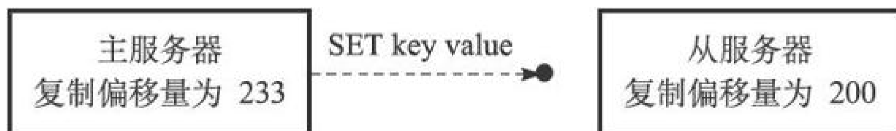


图15-24　主从服务器处于不一致状态

在这之后，当从服务器向主服务器发送REPLCONF ACK命令的时候，主服务器会察觉从服务器的复制偏移量依然为200，而自己的复制偏移量为233，这说明复制积压缓冲区里面复制偏移量为201至233的数

据（也即是命令SET key value）在传播过程中丢失了，于是主服务器会再次向从服务器传播命令SET key value，从服务器通过接收并执行这个命令可以将自己更新至主服务器当前所处的状态，如图15-25所示。

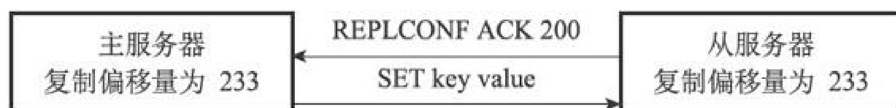


图15-25　主服务器向从服务器补发缺失的数据

注意，主服务器向从服务器补发缺失数据这一操作的原理和部分重同步操作的原理非常相似，这两个操作的区别在于，补发缺失数据操作在主从服务器没有断线的情况下执行，而部分重同步操作则在主从服务器断线并重连之后执行。

Redis 2.8版本以前的命令丢失

REPLCONF ACK命令和复制积压缓冲区都是Redis 2.8版本新增的，在Redis 2.8版本以前，即使命令在传播过程中丢失，主服务器和从服务器都不会注意到，主服务器更不会向从服务器补发丢失的数据，所以为了保证复制时主从服务器的数据一致性，最好使用2.8或以上版本的Redis。

15.8 重点回顾

·Redis 2.8以前的复制功能不能高效地处理断线后重复制情况，但Redis 2.8新添加的部分重同步功能可以解决这个问题。

·部分重同步通过复制偏移量、复制积压缓冲区、服务器运行ID三个部分来实现。

·在复制操作刚开始的时候，从服务器会成为主服务器的客户端，并通过向主服务器发送命令请求来执行复制步骤，而在复制操作的后期，主从服务器会互相成为对方的客户端。

·主服务器通过向从服务器传播命令来更新从服务器的状态，保持主从服务器一致，而从服务器则通过向主服务器发送命令来进行心跳检测，以及命令丢失检测。

第16章　Sentinel

Sentinel（哨岗、哨兵）是Redis的高可用性（high availability）解决方案：由一个或多个Sentinel实例（instance）组成的Sentinel系统（system）可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器，然后由新的主服务器代替已下线的主服务器继续处理命令请求。

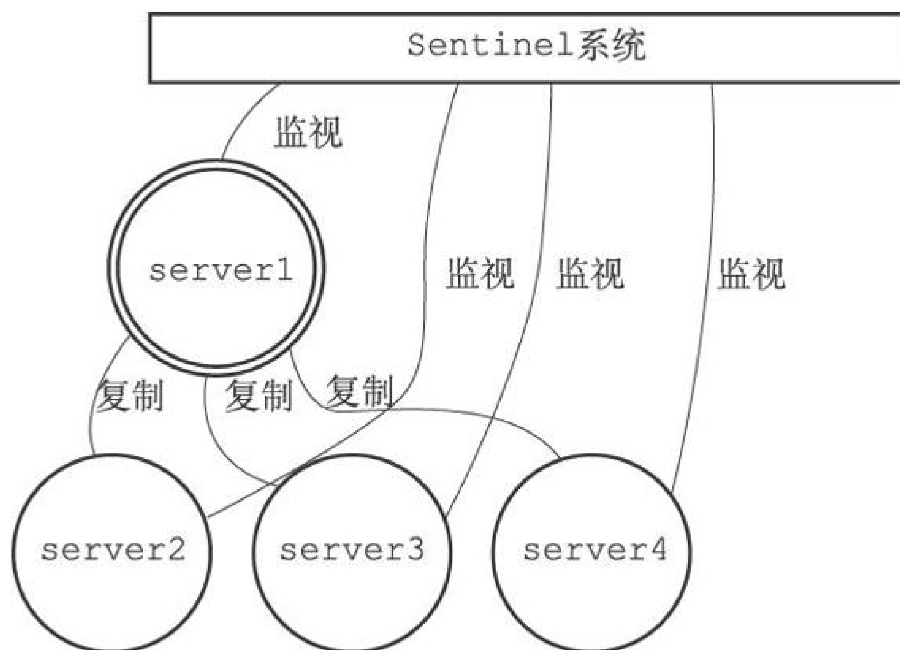


图16-1　服务器与Sentinel系统

图16-1展示了一个Sentinel系统监视服务器的例子，其中：

·用双环图案表示的是当前的主服务器server1。

·用单环图案表示的是主服务器的三个从服务器server2、server3以及server4。

·server2、server3、server4三个从服务器正在复制主服务器server1，而Sentinel系统则在监视所有四个服务器。

假设这时，主服务器server1进入下线状态，那么从服务器server2、server3、server4对主服务器的复制操作将被中止，并且Sentinel系统会察觉到server1已下线，如图16-2所示（下线的服务器用虚线表示）。

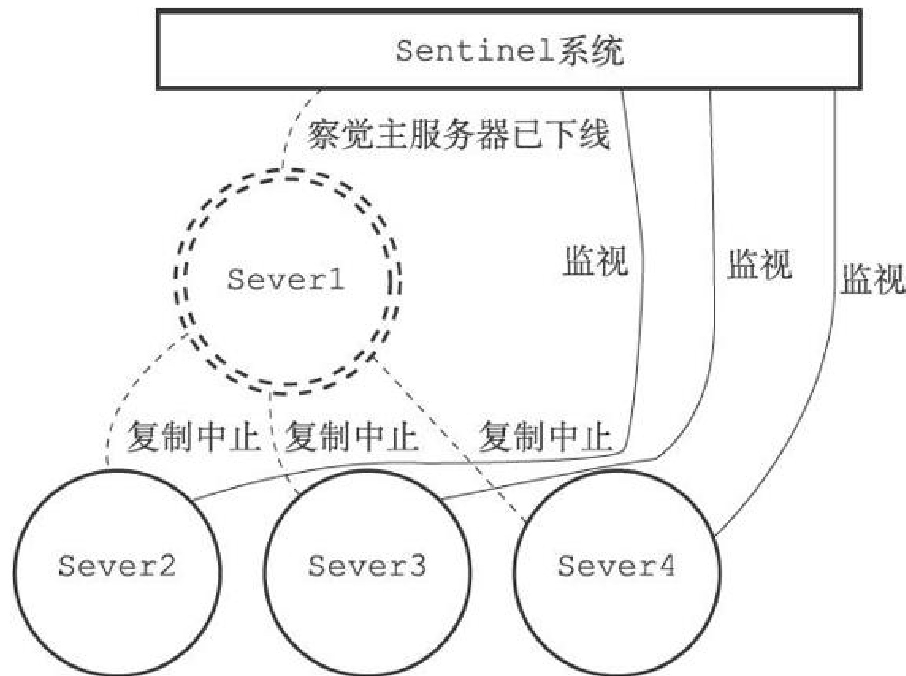


图16-2 主服务器下线

当server1的下线时长超过用户设定的下线时长上限时，Sentinel系统就会对server1执行故障转移操作：

·首先，Sentinel系统会挑选server1属下的其中一个从服务器，并将这个被选中的从服务器升级为新的主服务器。

·之后，Sentinel系统会向server1属下的所有从服务器发送新的复制指令，让它们成为新的主服务器的从服务器，当所有从服务器都开始复制新的主服务器时，故障转移操作执行完毕。

·另外，Sentinel还会继续监视已下线的server1，并在它重新上线时，将它设置为新的主服务器的从服务器。

举个例子，图16-3展示了Sentinel系统将server2升级为新的主服务器，并让服务器server3和server4成为server2的从服务器的过程。

之后，如果server1重新上线的话，它将被Sentinel系统降级为server2的从服务器，如图16-4所示。

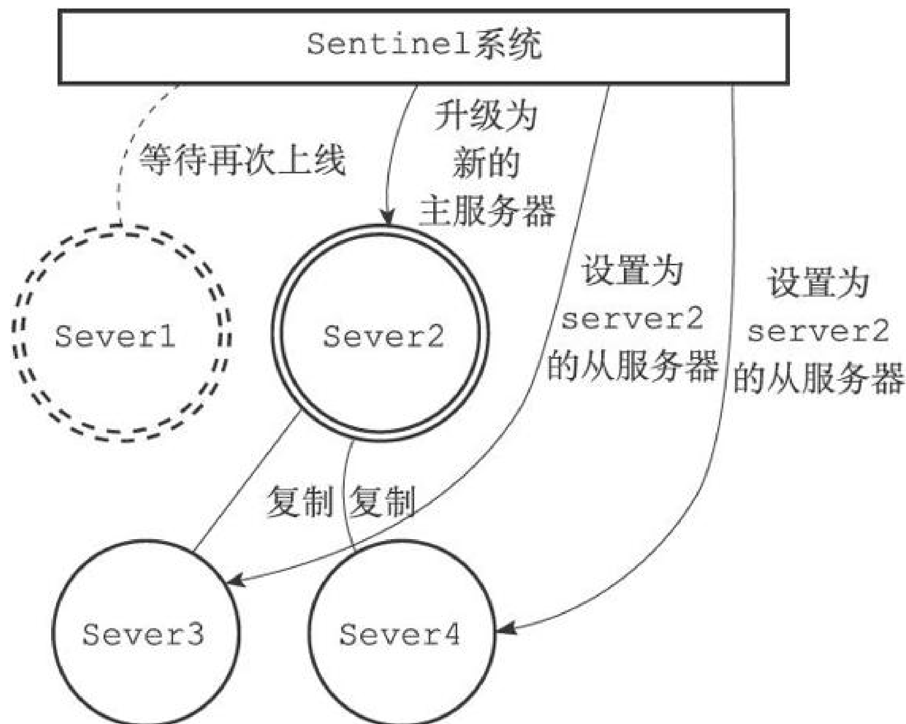


图16-3　故障转移

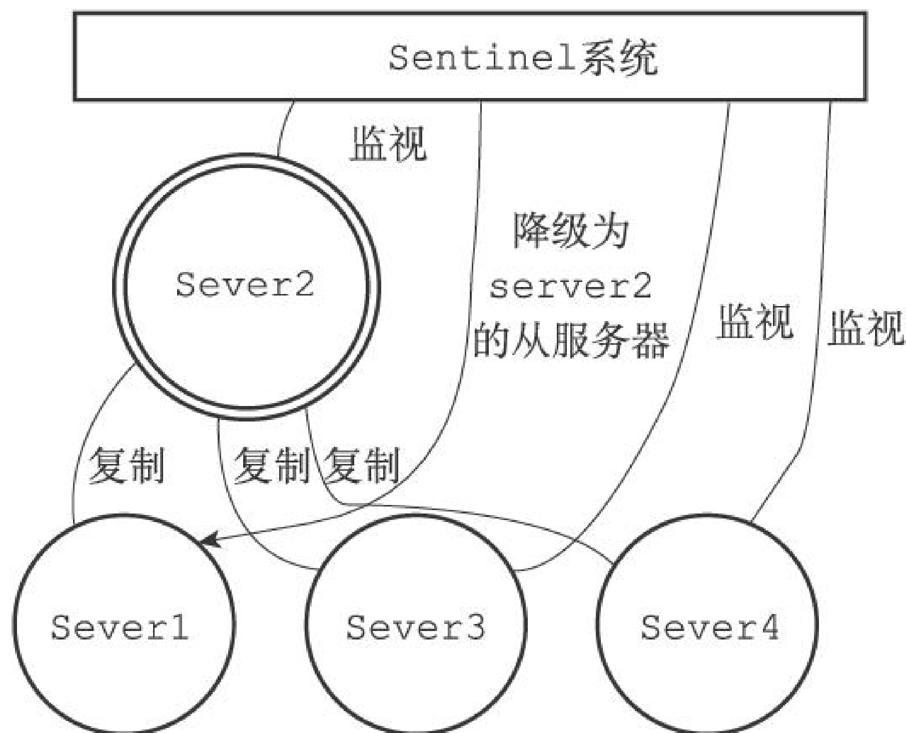


图16-4　原来的主服务器被降级为从服务器

本章首先会对Sentinel的初始化过程进行介绍，并说明Sentinel和一般Redis服务器的区别。

在此之后，本章将对Sentinel监视服务器的方法和原理进行介绍，说明Sentinel是如何判断一个服务器是否在线的。

最后，本章将介绍Sentinel系统对主服务器执行故障转移的整个过程。

16.1　启动并初始化Sentinel

启动一个Sentinel可以使用命令：

```
$ redis-sentinel /path/to/your/sentinel.conf
```

或者命令：

```
$ redis-server /path/to/your/sentinel.conf --sentinel
```

这两个命令的效果完全相同。

当一个Sentinel启动时，它需要执行以下步骤：

1）初始化服务器。

2）将普通Redis服务器使用的代码替换成Sentinel专用代码。

3）初始化Sentinel状态。

4）根据给定的配置文件，初始化Sentinel的监视主服务器列表。

5）创建连向主服务器的网络连接。

本节接下来的内容将分别对这些步骤进行介绍。

16.1.1　初始化服务器

首先，因为Sentinel本质上只是一个运行在特殊模式下的Redis服务器，所以启动Sentinel的第一步，就是初始化一个普通的Redis服务器，具体的步骤和第14章介绍的类似。

不过，因为Sentinel执行的工作和普通Redis服务器执行的工作不同，所以Sentinel的初始化过程和普通Redis服务器的初始化过程并不完全相同。

例如，普通服务器在初始化时会通过载入RDB文件或者AOF文件来还原数据库状态，但是因为Sentinel并不使用数据库，所以初始化Sentinel时就不会载入RDB文件或者AOF文件。

表16-1展示了Redis服务器在Sentinel模式下运行时，服务器各个主要功能的使用情况。

表16-1　Sentinel模式下Redis服务器主要功能的使用情况

功　　能	使用情况
数据库和键值对方面的命令，比如 *SET*、*DEL*、*FLUSHDB*	不使用
事务命令，比如 *MULTI* 和 *WATCH*	不使用
脚本命令，比如 *EVAL*	不使用
RDB 持久化命令，比如 *SAVE* 和 *BGSAVE*	不使用
AOF 持久化命令，比如 *BGREWRITEAOF*	不使用
复制命令，比如 *SLAVEOF*	Sentinel 内部可以使用，但客户端不可以使用
发布与订阅命令，比如 *PUBLISH* 和 *SUBSCRIBE*	*SUBSCRIBE*、*PSUBSCRIBE*、*UNSUBSCRIBE*、*PUNSUBSCRIBE* 四个命令在 Sentinel 内部和客户端都可以使用，但 *PUBLISH* 命令只能在 Sentinel 内部使用
文件事件处理器（负责发送命令请求、处理命令回复）	Sentinel 内部使用，但关联的文件事件处理器和普通 Redis 服务器不同
时间事件处理器（负责执行 serverCron 函数）	Sentinel 内部使用，时间事件的处理器仍然是 serverCron 函数，serverCron 函数会调用 sentinel.c/sentinelTimer 函数，后者包含了 Sentinel 要执行的所有操作

16.1.2　使用Sentinel专用代码

启动Sentinel的第二个步骤就是将一部分普通Redis服务器使用的代码替换成Sentinel专用代码。比如说，普通Redis服务器使用redis.h/REDIS_SERVERPORT常量的值作为服务器端口：

```
#define REDIS_SERVERPORT 6379
```

而Sentinel则使用sentinel.c/REDIS_SENTINEL_PORT常量的值作为服务器端口：

```
#define REDIS_SENTINEL_PORT 26379
```

除此之外，普通Redis服务器使用redis.c/redisCommandTable作为服务器的命令表：

```
struct redisCommand redisCommandTable[] = {
    {"get",getCommand,2,"r",0,NULL,1,1,1,0,0},
    {"set",setCommand,-3,"wm",0,noPreloadGetKeys,1,1,1,0,0},
    {"setnx",setnxCommand,3,"wm",0,noPreloadGetKeys,1,1,1,0,0},
    // ...
    {"script",scriptCommand,-2,"ras",0,NULL,0,0,0,0,0},
    {"time",timeCommand,1,"rR",0,NULL,0,0,0,0,0},
    {"bitop",bitopCommand,-4,"wm",0,NULL,2,-1,1,0,0},
    {"bitcount",bitcountCommand,-2,"r",0,NULL,1,1,1,0,0}
}
```

而Sentinel则使用sentinel.c/sentinelcmds作为服务器的命令表，并且其中的INFO命令会使用Sentinel模式下的专用实现sentinel.c/sentinelInfoCommand函数，而不是普通Redis服务器使用的实现redis.c/infoCommand函数：

```
struct redisCommand sentinelcmds[] = {
    {"ping",pingCommand,1,"",0,NULL,0,0,0,0,0},
    {"sentinel",sentinelCommand,-2,"",0,NULL,0,0,0,0,0},
    {"subscribe",subscribeCommand,-2,"",0,NULL,0,0,0,0,0},
    {"unsubscribe",unsubscribeCommand,-1,"",0,NULL,0,0,0,0,0},
    {"psubscribe",psubscribeCommand,-2,"",0,NULL,0,0,0,0,0},
    {"punsubscribe",punsubscribeCommand,-1,"",0,NULL,0,0,0,0,0},
    {"info",sentinelInfoCommand,-1,"",0,NULL,0,0,0,0,0}
};
```

sentinelcmds命令表也解释了为什么在Sentinel模式下，Redis服务器不能执行诸如SET、DBSIZE、EVAL等等这些命令，因为服务器根本没有在命令表中载入这些命令。PING、SENTINEL、INFO、SUBSCRIBE、UNSUBSCRIBE、PSUBSCRIBE和PUNSUBSCRIBE这七个命令就是客户端可以对Sentinel执行的全部命令了。

16.1.3 初始化Sentinel状态

在应用了Sentinel的专用代码之后，接下来，服务器会初始化一个sentinel.c/sentinelState结构（后面简称“Sentinel状态”），这个结构保存了服务器中所有和Sentinel功能有关的状态（服务器的一般状态仍然由redis.h/redisServer结构保存）：

```
struct sentinelState {
    //
当前纪元，用于实现故障转移
    uint64_t current_epoch;
    //
保存了所有被这个sentinel
监视的主服务器
    //
字典的键是主服务器的名字
    //
字典的值则是一个指向sentinelRedisInstance
结构的指针
    dict *masters;
    //
是否进入了TILT
模式？
    int tilt;
    //
目前正在执行的脚本的数量
    int running_scripts;
    //
进入TILT
模式的时间
    mstime_t tilt_start_time;
    //
最后一次执行时间处理器的时间
    mstime_t previous_time;
    //
一个FIFO
队列，包含了所有需要执行的用户脚本
    list *scripts_queue;
} sentinel;
```

16.1.4 初始化Sentinel状态的masters属性

Sentinel状态中的masters字典记录了所有被Sentinel监视的主服务

器的相关信息，其中：

·字典的键是被监视主服务器的名字。

·而字典的值则是被监视主服务器对应的sentinel.c/sentinelRedisInstance结构。

每个sentinelRedisInstance结构（后面简称“实例结构”）代表一个被Sentinel监视的Redis服务器实例（instance），这个实例可以是主服务器、从服务器，或者另外一个Sentinel。

实例结构包含的属性非常多，以下代码展示了实例结构在表示主服务器时使用的其中一部分属性，本章接下来将逐步对实例结构中的各个属性进行介绍：

```
typedef struct sentinelRedisInstance {
    //
标识值，记录了实例的类型，以及该实例的当前状态
    int flags;
    //
实例的名字
    //
主服务器的名字由用户在配置文件中设置
    //
从服务器以及Sentinel
的名字由Sentinel
自动设置
    //
格式为ip:port
，例如"127.0.0.1:26379"
    char *name;
    //
实例的运行ID
    char *runid;
    //
配置纪元，用于实现故障转移
    uint64_t config_epoch;
    //
实例的地址
    sentinelAddr *addr;
    // SENTINEL down-after-milliseconds
选项设定的值
    //
实例无响应多少毫秒之后才会被判断为主观下线（subjectively down
）
    mstime_t down_after_period;
    // SENTINEL monitor <master-name> <IP> <port> <quorum>
选项中的quorum
参数
    //
判断这个实例为客观下线（objectively down
）所需的支持投票数量
    int quorum;
    // SENTINEL parallel-syncs <master-name> <number>
选项的值
    //
在执行故障转移操作时，可以同时对新的主服务器进行同步的从服务器数量
    int parallel_syncs;
    // SENTINEL failover-timeout <master-name> <ms>
选项的值
    //
刷新故障迁移状态的最大时限
```

```
    mstime_t failover_timeout;
    // ...
} sentinelRedisInstance;
```

sentinelRedisInstance.addr属性是一个指向sentinel.c/sentinelAddr结构的指针，这个结构保存着实例的IP地址和端口号：

```
typedef struct sentinelAddr {
    char *ip;
    int port;
} sentinelAddr;
```

对Sentinel状态的初始化将引发对masters字典的初始化，而masters字典的初始化是根据被载入的Sentinel配置文件来进行的。

举个例子，如果用户在启动Sentinel时，指定了包含以下内容的配置文件：

```
#####################
# master1 configure #
#####################
sentinel monitor master1 127.0.0.1 6379 2
sentinel down-after-milliseconds master1 30000
sentinel parallel-syncs master1 1
sentinel failover-timeout master1 900000
#####################
# master2 configure #
#####################
sentinel monitor master2 127.0.0.1 12345 5
sentinel down-after-milliseconds master2 50000
sentinel parallel-syncs master2 5
sentinel failover-timeout master2 450000
```

那么Sentinel将为主服务器master1创建如图16-5所示的实例结构，并为主服务器master2创建如图16-6所示的实例结构，而这两个实例结构又会被保存到Sentinel状态的masters字典中，如图16-7所示。

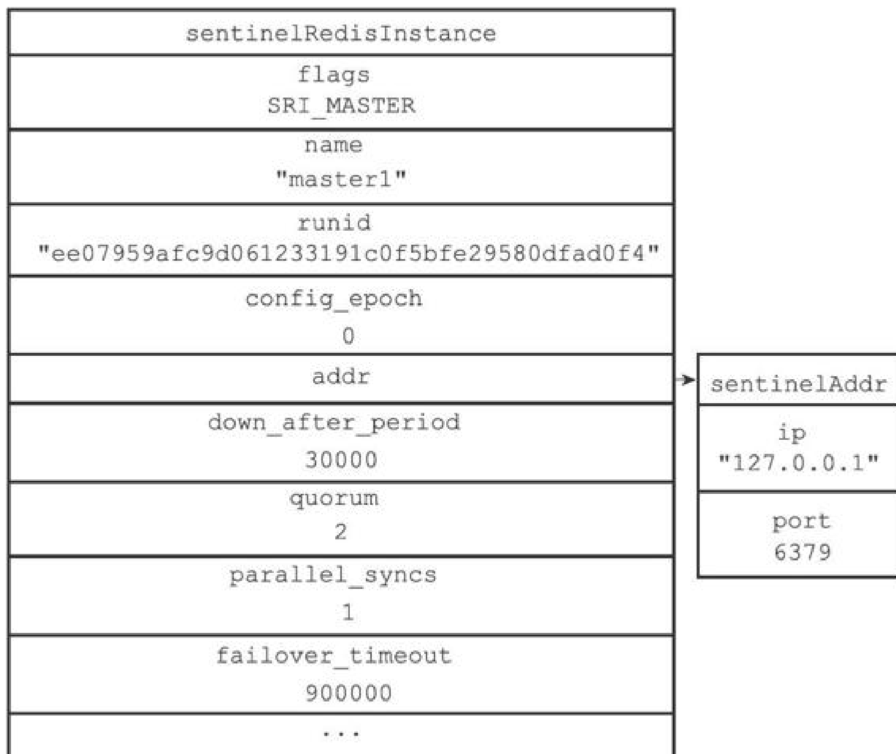


图16-5　master1的实例结构

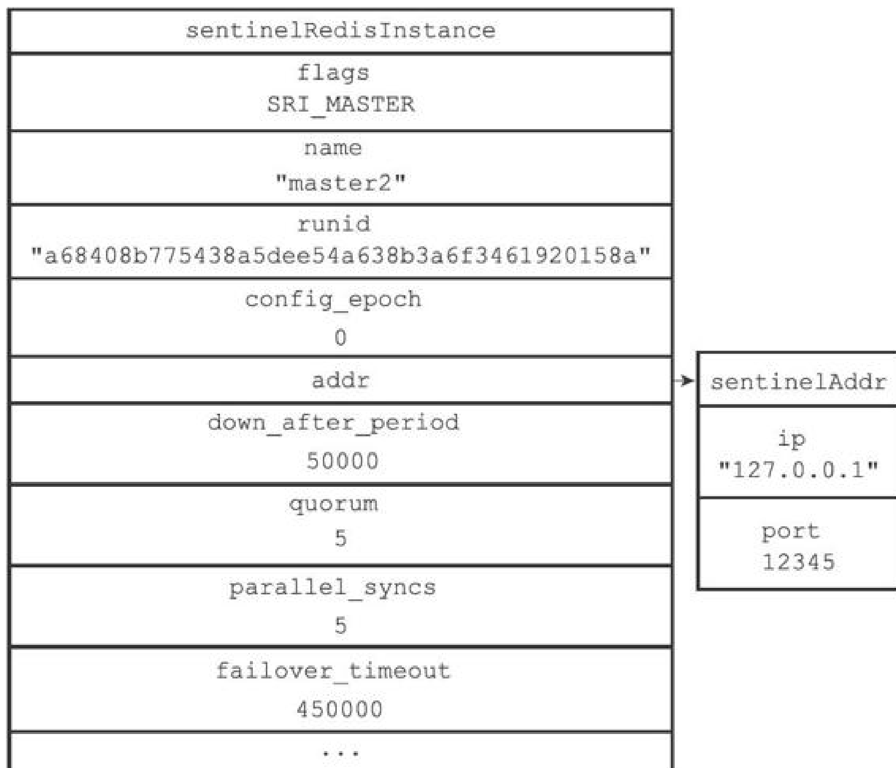


图16-6　master2的实例结构

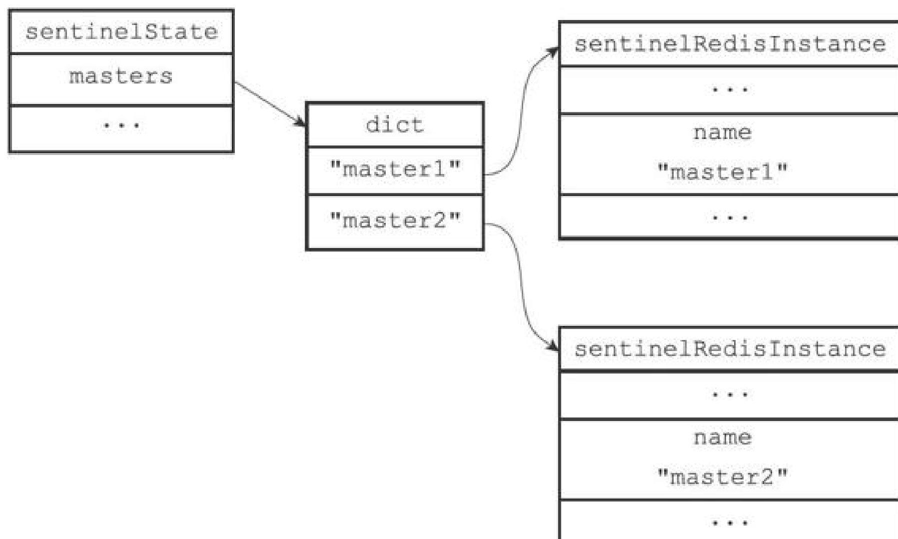


图16-7　Sentinel状态以及masters字典

16.1.5　创建连向主服务器的网络连接

初始化Sentinel的最后一步是创建连向被监视主服务器的网络连接，Sentinel将成为主服务器的客户端，它可以向主服务器发送命令，并从命令回复中获取相关的信息。

对于每个被Sentinel监视的主服务器来说，Sentinel会创建两个连向主服务器的异步网络连接：

·一个是命令连接，这个连接专门用于向主服务器发送命令，并接收命令回复。

·另一个是订阅连接，这个连接专门用于订阅主服务器的__sentinel__:hello频道。

为什么有两个连接？

在Redis目前的发布与订阅功能中，被发送的信息都不会保存在Redis服务器里面，如果在信息发送时，想要接收信息的客户端不在线或者断线，那么这个客户端就会丢失这条信息。因此，为了不丢失__sentinel__:hello频道的任何信息，Sentinel必须专门用一个订阅连接来接收该频道的信息。

另一方面，除了订阅频道之外，Sentinel还必须向主服务器发送命令，以此来与主服务器进行通信，所以Sentinel还必须向主服

务器创建命令连接。

因为Sentinel需要与多个实例创建多个网络连接，所以Sentinel使用的是异步连接。

图16-8展示了一个Sentinel向被它监视的两个主服务器master1和master2创建命令连接和订阅连接的例子。

接下来的一节将介绍Sentinel是如何通过命令连接和订阅连接来与被监视主服务器进行通信的。

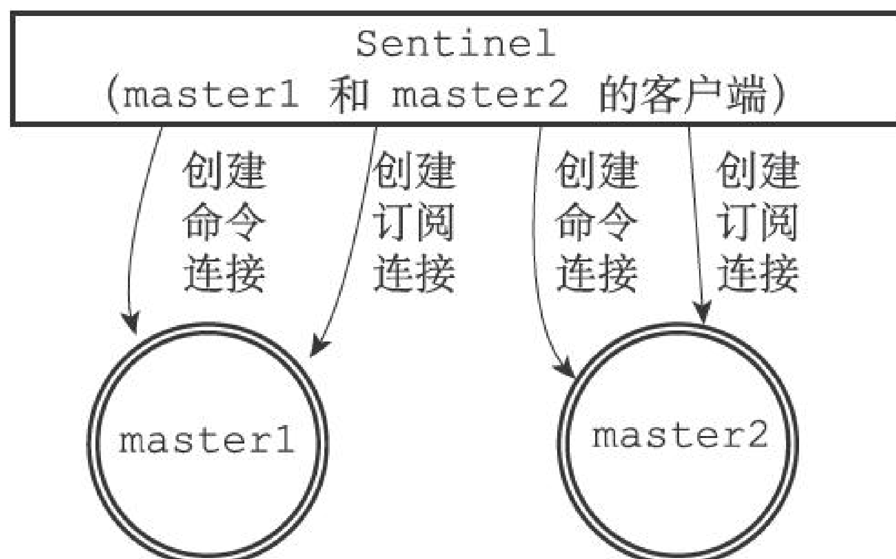


图16-8　Sentinel向主服务器创建网络连接

16.2　获取主服务器信息

Sentinel默认会以每十秒一次的频率，通过命令连接向被监视的主服务器发送INFO命令，并通过分析INFO命令的回复来获取主服务器的当前信息。

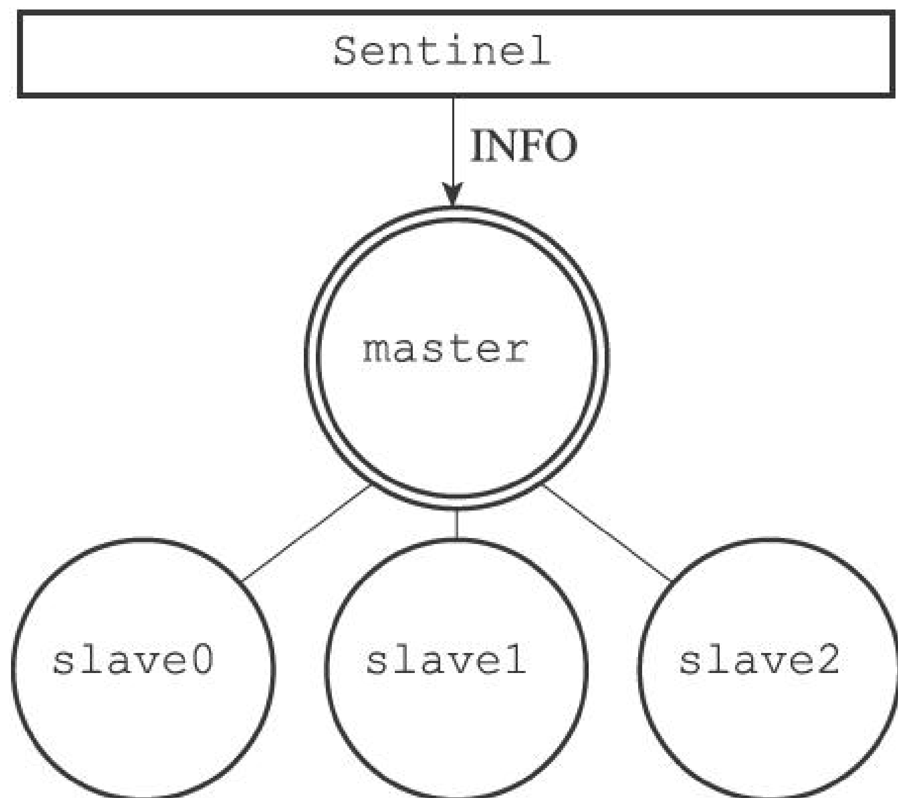


图16-9　Sentinel向带有三个从服务器的主服务器发送INFO命令

举个例子，假设如图16-9所示，主服务器master有三个从服务器slave0、slave1和slave2，并且一个Sentinel正在连接主服务器，那么Sentinel将持续地向主服务器发送INFO命令，并获得类似于以下内容的回复：

```
# Server
...
run_id:7611c59dc3a29aa6fa0609f841bb6a1019008a9c
...
# Replication
role:master
...
slave0:ip=127.0.0.1,port=11111,state=online,offset=43,lag=0
slave1:ip=127.0.0.1,port=22222,state=online,offset=43,lag=0
slave2:ip=127.0.0.1,port=33333,state=online,offset=43,lag=0
...
# Other sections
```

...

通过分析主服务器返回的INFO命令回复，Sentinel可以获取以下两方面的信息：

·一方面是关于主服务器本身的信息，包括run_id域记录的服务器运行ID，以及role域记录的服务器角色；

·另一方面是关于主服务器属下所有从服务器的信息，每个从服务器都由一个"slave"字符串开头的行记录，每行的ip＝域记录了从服务器的IP地址，而port＝域则记录了从服务器的端口号。根据这些IP地址和端口号，Sentinel无须用户提供从服务器的地址信息，就可以自动发现从服务器。

根据run_id域和role域记录的信息，Sentinel将对主服务器的实例结构进行更新，例如，主服务器重启之后，它的运行ID就会和实例结构之前保存的运行ID不同，Sentinel检测到这一情况之后，就会对实例结构的运行ID进行更新。

至于主服务器返回的从服务器信息，则会被用于更新主服务器实例结构的slaves字典，这个字典记录了主服务器属下从服务器的名单：

·字典的键是由Sentinel自动设置的从服务器名字，格式为ip:port：如对于IP地址为127.0.0.1，端口号为11111的从服务器来说，Sentinel为它设置的名字就是127.0.0.1:11111。

·至于字典的值则是从服务器对应的实例结构：比如说，如果键是127.0.0.1:11111，那么这个键的值就是IP地址为127.0.0.1，端口号为11111的从服务器的实例结构。

Sentinel在分析INFO命令中包含的从服务器信息时，会检查从服务器对应的实例结构是否已经存在于slaves字典：

·如果从服务器对应的实例结构已经存在，那么Sentinel对从服务器的实例结构进行更新。

·如果从服务器对应的实例结构不存在，那么说明这个从服务器是新发现的从服务器，Sentinel会在slaves字典中为这个从服务器新创建一个实例结构。

对于我们之前列举的主服务器master和三个从服务器slave0、slave1和slave2的例子来说，Sentinel将分别为三个从服务器创建它们各自的实例结构，并将这些结构保存到主服务器实例结构的slaves字典里面，如图16-10所示。

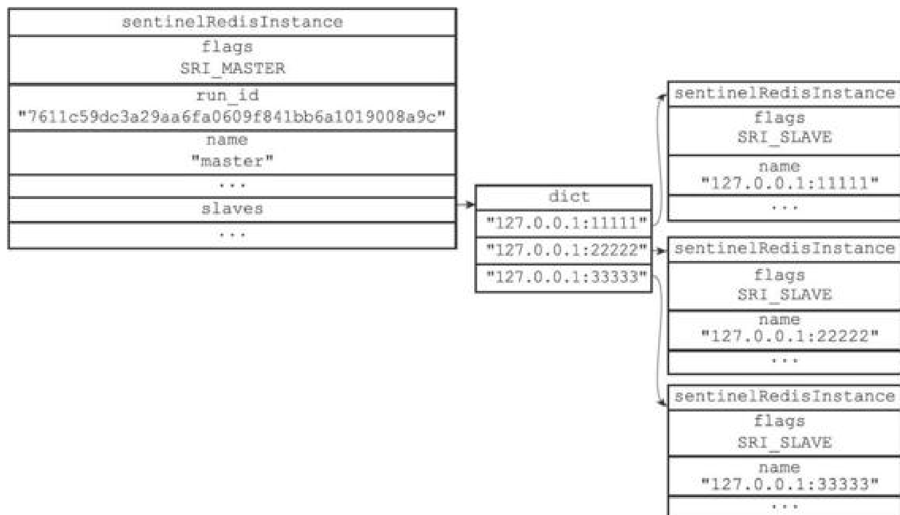


图16-10　主服务器和它的三个从服务器

注意对比图中主服务器实例结构和从服务器实例结构之间的区别：

·主服务器实例结构的flags属性的值为SRI_MASTER，而从服务器实例结构的flags属性的值为SRI_SLAVE。

·主服务器实例结构的name属性的值是用户使用Sentinel配置文件设置的，而从服务器实例结构的name属性的值则是Sentinel根据从服务器的IP地址和端口号自动设置的。

16.3 获取从服务器信息

当Sentinel发现主服务器有新的从服务器出现时，Sentinel除了会为这个新的从服务器创建相应的实例结构之外，Sentinel还会创建连接到从服务器的命令连接和订阅连接。

举个例子，对于图16-10所示的主从服务器关系来说，Sentinel将对slave0、slave1和slave2三个从服务器分别创建命令连接和订阅连接，如图16-11所示。

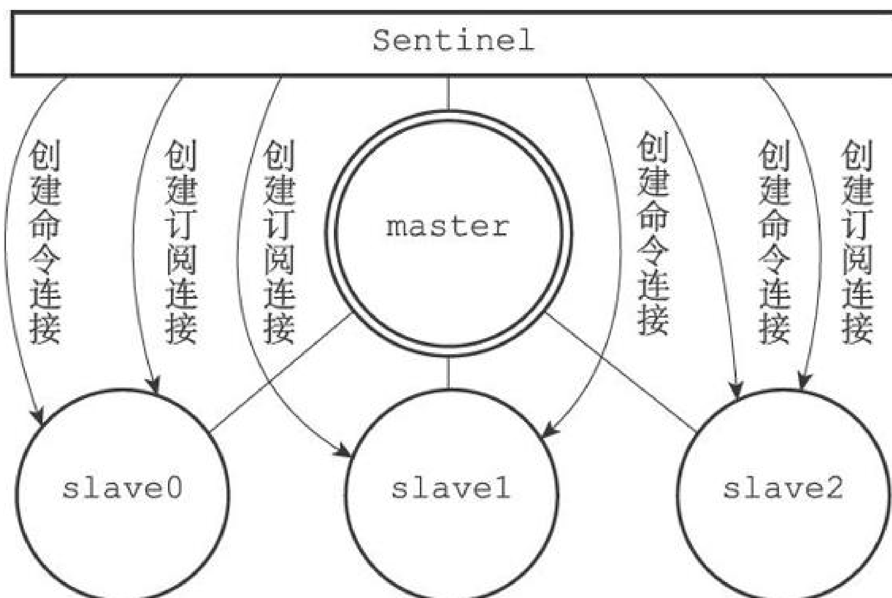


图16-11　Sentinel与各个从服务器建立命令连接和订阅连接

在创建命令连接之后，Sentinel在默认情况下，会以每十秒一次的频率通过命令连接向从服务器发送INFO命令，并获得类似于以下内容的回复：

```
# Server
...
run_id:32be0699dd27b410f7c90dada3a6fab17f97899f
...
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
slave_repl_offset:11887
slave_priority:100
# Other sections
...
```

根据INFO命令的回复，Sentinel会提取出以下信息：

·从服务器的运行ID run_id。

·从服务器的角色role。

·主服务器的IP地址master_host，以及主服务器的端口号master_port。

·主从服务器的连接状态master_link_status。

·从服务器的优先级slave_priority。

·从服务器的复制偏移量slave_repl_offset。

根据这些信息，Sentinel会对从服务器的实例结构进行更新，图16-12展示了Sentinel根据上面的INFO命令回复对从服务器的实例结构进行更新之后，实例结构的样子。

sentinelRedisInstance
flags SRI_SLAVE
run_id "32be0699dd27b410f7c90dada3a6fab17f97899f"
slave_master_host "127.0.0.1"
slave_master_port 6379
slave_master_link_status SENTINEL_MASTER_LINK_STATUS_UP
slave_repl_offset 11887
slave_priority 100
...

图16-12　从服务器实例结构

16.4　向主服务器和从服务器发送信息

在默认情况下，Sentinel会以每两秒一次的频率，通过命令连接向所有被监视的主服务器和从服务器发送以下格式的命令：

```
PUBLISH __sentinel__:hello "<s_ip>,<s_port>,<s_runid>,<s_epoch>,<m_name>,<m_ip>,<m_po
```

这条命令向服务器的_sentinel_:hello频道发送了一条信息，信息的内容由多个参数组成：

·其中以s_开头的参数记录的是Sentinel本身的信息，各个参数的意义如表16-2所示。

·而m_开头的参数记录的则是主服务器的信息，各个参数的意义如表16-3所示。如果Sentinel正在监视的是主服务器，那么这些参数记录的就是主服务器的信息；如果Sentinel正在监视的是从服务器，那么这些参数记录的就是从服务器正在复制的主服务器的信息。

表16-2　信息中和Sentinel有关的参数

参　数	意　义
s_ip	Sentinel 的 IP 地址
s_port	Sentinel 的端口号
s_runid	Sentinel 的运行 ID
s_epoch	Sentinel 当前的配置纪元（configuration epoch）

表16-3　信息中和主服务器有关的参数

参　数	意　义
m_name	主服务器的名字
m_ip	主服务器的 IP 地址
m_port	主服务器的端口号
m_epoch	主服务器当前的配置纪元

以下是一条Sentinel通过PUBLISH命令向主服务器发送的信息示例：

```
"127.0.0.1,26379,e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa,0,mymaster,127.0.0.1,6379,0
```

这个示例包含了以下信息：

·Sentinel的IP地址为127.0.0.1端口号为26379，运行ID为e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa，当前的配置纪元为0。

·主服务器的名字为mymaster，IP地址为127.0.0.1，端口号为

6379，当前的配置纪元为0。

16.5 接收来自主服务器和从服务器的频道信息

当Sentinel与一个主服务器或者从服务器建立起订阅连接之后，Sentinel就会通过订阅连接，向服务器发送以下命令：

```
SUBSCRIBE __sentinel__:hello
```

Sentinel对__sentinel__:hello频道的订阅会一直持续到Sentinel与服务器的连接断开为止。

这也就是说，对于每个与Sentinel连接的服务器，Sentinel既通过命令连接向服务器的__sentinel__:hello频道发送信息，又通过订阅连接从服务器的__sentinel__:hello频道接收信息，如图16-13所示。

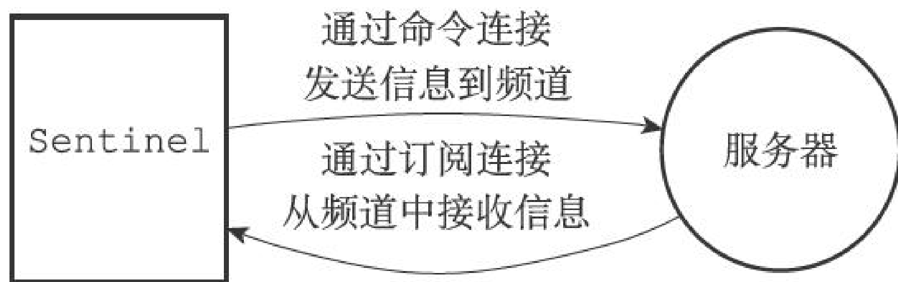


图16-13　Sentinel同时向服务器发送和接收信息

对于监视同一个服务器的多个Sentinel来说，一个Sentinel发送的信息会被其他Sentinel接收到，这些信息会被用于更新其他Sentinel对发送信息Sentinel的认知，也会被用于更新其他Sentinel对被监视服务器的认知。

举个例子，假设现在有sentinel1、sentinel2、sentinel3三个Sentinel在监视同一个服务器，那么当sentinel1向服务器的__sentinel__:hello频道发送一条信息时，所有订阅了__sentinel__:hello频道的Sentinel（包括sentinel1自己在内）都会收到这条信息，如图16-14所示。

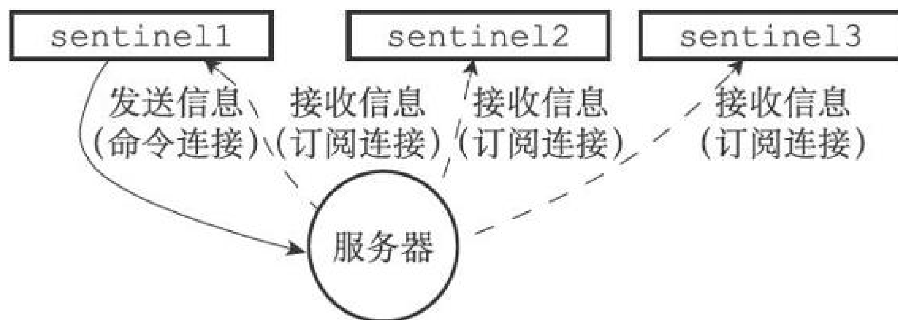


图16-14　向服务器发送信息

当一个Sentinel从_sentinel_:hello频道收到一条信息时，Sentinel会对这条信息进行分析，提取出信息中的Sentinel　IP地址、Sentinel端口号、Sentinel运行ID等八个参数，并进行以下检查：

·如果信息中记录的Sentinel运行ID和接收信息的Sentinel的运行ID相同，那么说明这条信息是Sentinel自己发送的，Sentinel将丢弃这条信息，不做进一步处理。

·相反地，如果信息中记录的Sentinel运行ID和接收信息的Sentinel的运行ID不相同，那么说明这条信息是监视同一个服务器的其他Sentinel发来的，接收信息的Sentinel将根据信息中的各个参数，对相应主服务器的实例结构进行更新。

16.5.1　更新sentinels字典

Sentinel为主服务器创建的实例结构中的sentinels字典保存了除Sentinel本身之外，所有同样监视这个主服务器的其他Sentinel的资料：

·sentinels字典的键是其中一个Sentinel的名字，格式为ip:port，比如对于IP地址为127.0.0.1，端口号为26379的Sentinel来说，这个Sentinel在sentinels字典中的键就是"127.0.0.1:26379"。

·sentinels字典的值则是键所对应Sentinel的实例结构，比如对于键"127.0.0.1:26379"来说，这个键在sentinels字典中的值就是IP为127.0.0.1，端口号为26379的Sentinel的实例结构。

当一个Sentinel接收到其他Sentinel发来的信息时（我们称呼发送信息的Sentinel为源Sentinel，接收信息的Sentinel为目标Sentinel），目标Sentinel会从信息中分析并提取出以下两方面参数：

·与Sentinel有关的参数：源Sentinel的IP地址、端口号、运行ID和配置纪元。

·与主服务器有关的参数：源Sentinel正在监视的主服务器的名字、IP地址、端口号和配置纪元。

根据信息中提取出的主服务器参数，目标Sentinel会在自己的Sentinel状态的masters字典中查找相应的主服务器实例结构，然后根据提取出的Sentinel参数，检查主服务器实例结构的sentinels字典中，源Sentinel的实例结构是否存在：

·如果源Sentinel的实例结构已经存在，那么对源Sentinel的实例结构进行更新。

·如果源Sentinel的实例结构不存在，那么说明源Sentinel是刚刚开始监视主服务器的新Sentinel，目标Sentinel会为源Sentinel创建一个新的实例结构，并将这个结构添加到sentinels字典里面。

举个例子，假设分别有127.0.0.1:26379、127.0.0.1:26380、127.0.0.1:26381三个Sentinel正在监视主服务器127.0.0.1:6379，那么当127.0.0.1:26379这个Sentinel接收到以下信息时：

```
1) "message"
2) "__sentinel__:hello"
3) "127.0.0.1,26379,e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa,0,mymaster,127.0.0.1,637
1) "message"
2) "__sentinel__:hello"
3) "127.0.0.1,26381,6241bf5cf9bfc8ecd15d6eb6cc3185edfbb24903,0,mymaster,127.0.0.1,637
1) "message"
2) "__sentinel__:hello"
3) "127.0.0.1,26380,a9b22fb79ae8fad28e4ea77d20398f77f6b89377,0,mymaster,127.0.0.1,637
```

Sentinel将执行以下动作：

·第一条信息的发送者为127.0.0.1:26379自己，这条信息会被忽略。

·第二条信息的发送者为127.0.0.1:26381，Sentinel会根据这条信息中提取出的内容，对sentinels字典中127.0.0.1:26381对应的实例结构进行更新。

·第三条信息的发送者为127.0.0.1:26380，Sentinel会根据这条信息中提取出的内容，对sentinels字典中127.0.0.1:26380所对应的实例结构进行更新。

图16-15展示了Sentinel 127.0.0.1:26379为主服务器127.0.0.1:6379创建的实例结构，以及结构中的sentinels字典。

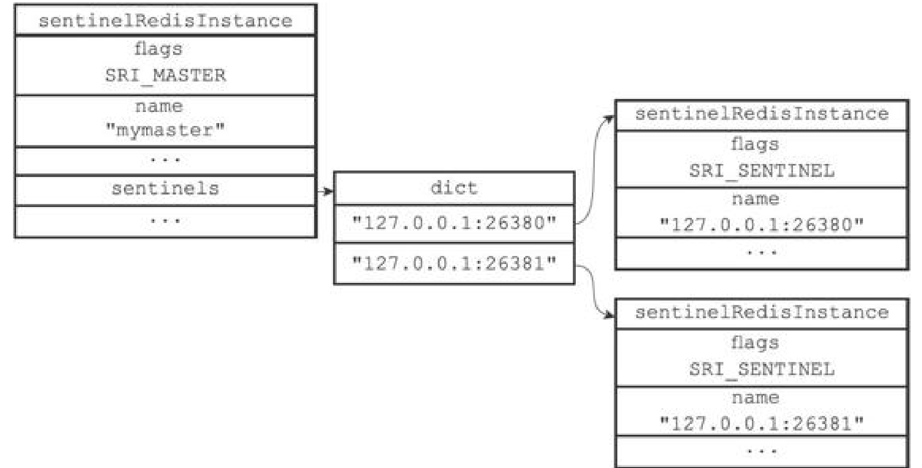


图16-15　主服务器实例结构中的sentinels字典

和127.0.0.1:26379一样，其他两个Sentinel也会创建类似于图16-15所示的sentinels字典，区别在于字典中保存的Sentinel信息不同：

·127.0.0.1:26380创建的sentinels字典会保存127.0.0.1:26379和127.0.0.1:26381两个Sentinel的信息。

·而127.0.0.1:26381创建的sentinels字典则会保存127.0.0.1:26379和127.0.0.1:26380两个Sentinel的信息。

因为一个Sentinel可以通过分析接收到的频道信息来获知其他Sentinel的存在，并通过发送频道信息来让其他Sentinel知道自己的存在，所以用户在使用Sentinel的时候并不需要提供各个Sentinel的地址信息，监视同一个主服务器的多个Sentinel可以自动发现对方。

16.5.2 创建连向其他Sentinel的命令连接

当Sentinel通过频道信息发现一个新的Sentinel时，它不仅会为新Sentinel在sentinels字典中创建相应的实例结构，还会创建一个连向新Sentinel的命令连接，而新Sentinel也同样会创建连向这个Sentinel的命令连接，最终监视同一主服务器的多个Sentinel将形成相互连接的网络：Sentinel A有连向Sentinel B的命令连接，而Sentinel B也有连向Sentinel A的命令连接。

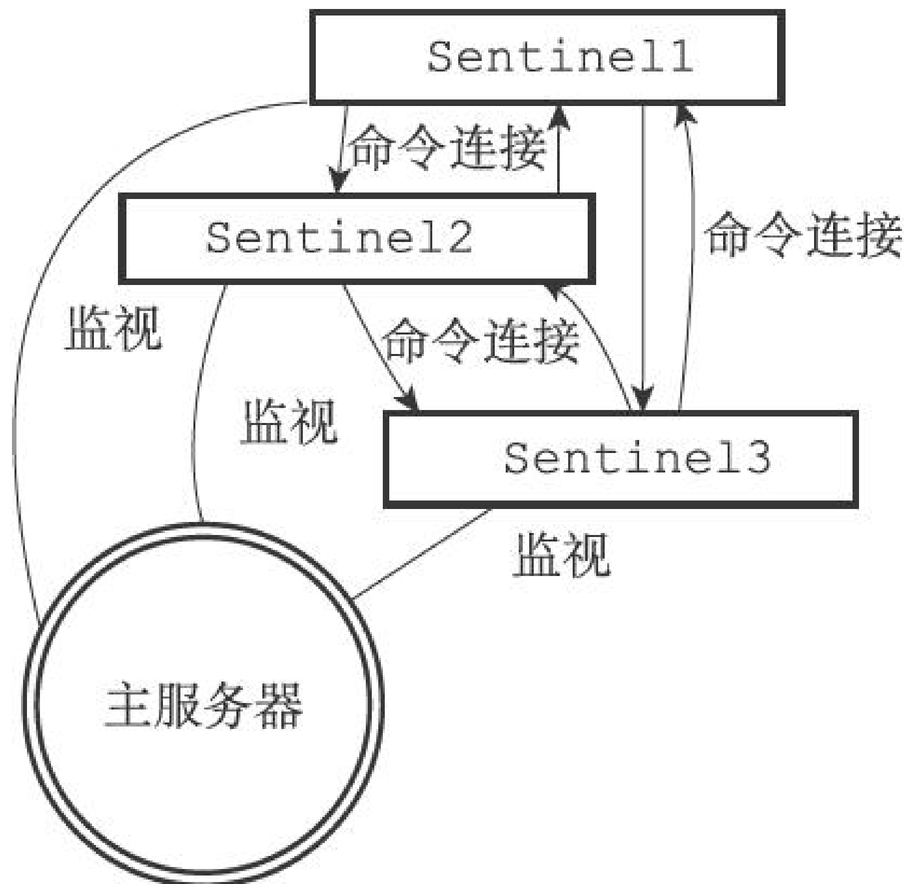


图16-16　各个Sentinel之间的网络连接

图16-16展示了三个监视同一主服务器的Sentinel之间是如何互相连接的。

使用命令连接相连的各个Sentinel可以通过向其他Sentinel发送命令请求来进行信息交换，本章接下来将对Sentinel实现主观下线检测和客观下线检测的原理进行介绍，这两种检测都会使用Sentinel之间的命令连接来进行通信。

Sentinel之间不会创建订阅连接

Sentinel在连接主服务器或者从服务器时，会同时创建命令连接和订阅连接，但是在连接其他Sentinel时，却只会创建命令连接，而不创建订阅连接。这是因为Sentinel需要通过接收主服务器或者从服务器发来的频道信息来发现未知的新Sentinel，所以才需要建立订阅连接，而相互已知的Sentinel只要使用命令连接来进行通信就足够了。

16.6　检测主观下线状态

在默认情况下，Sentinel会以每秒一次的频率向所有与它创建了命令连接的实例（包括主服务器、从服务器、其他Sentinel在内）发送PING命令，并通过实例返回的PING命令回复来判断实例是否在线。

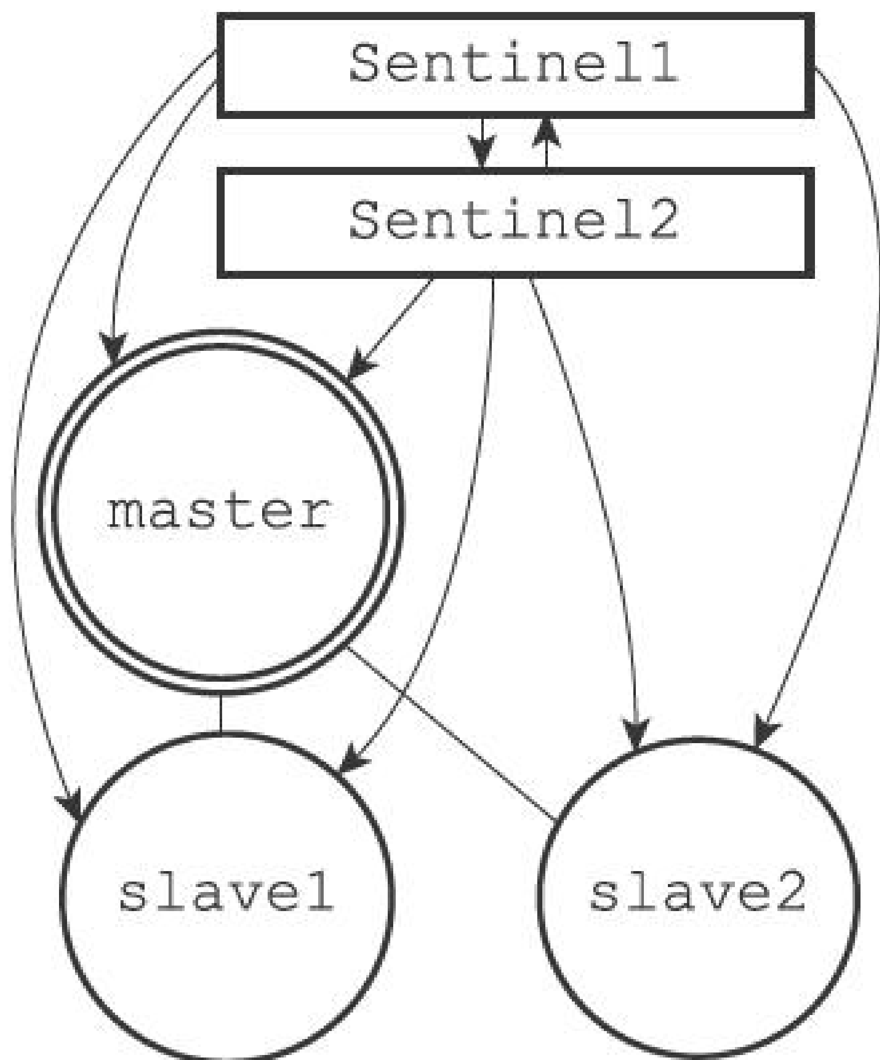


图16-17　Sentinel向实例发送PING命令

在图16-17展示的例子中，带箭头的连线显示了Sentinel1和Sentinel2是如何向实例发送PING命令的：

·Sentinel1将向Sentinel2、主服务器master、从服务器slave1和slave2发送PING命令。

·Sentinel2将向Sentinel1、主服务器master、从服务器slave1和slave2发送PING命令。

实例对PING命令的回复可以分为以下两种情况：

·有效回复：实例返回＋PONG、-LOADING、-MASTERDOWN三种回复的其中一种。

·无效回复：实例返回除＋PONG、-LOADING、-MASTERDOWN三种回复之外的其他回复，或者在指定时限内没有返回任何回复。

Sentinel配置文件中的down-after-milliseconds选项指定了Sentinel判断实例进入主观下线所需的时间长度：如果一个实例在down-after-milliseconds毫秒内，连续向Sentinel返回无效回复，那么Sentinel会修改这个实例所对应的实例结构，在结构的flags属性中打开SRI_S_DOWN标识，以此来表示这个实例已经进入主观下线状态。

以图16-17展示的情况为例子，如果配置文件指定Sentinel1的down-after-milliseconds选项的值为50000毫秒，那么当主服务器master连续50000毫秒都向Sentinel1返回无效回复时，Sentinel1就会将master标记为主观下线，并在master所对应的实例结构的flags属性中打开SRI_S_DOWN标识，如图16-18所示。



图16-18　主服务器被标记为主观下线

主观下线时长选项的作用范围

用户设置的down-after-milliseconds选项的值，不仅会被Sentinel用来判断主服务器的主观下线状态，还会被用于判断主服

务器属下的所有从服务器，以及所有同样监视这个主服务器的其他Sentinel的主观下线状态。举个例子，如果用户向Sentinel设置了以下配置：

```
sentinel monitor master 127.0.0.1 6379 2
sentinel down-after-milliseconds master 50000
```

那么50000毫秒不仅会成为Sentinel判断master进入主观下线的标准，还会成为Sentinel判断master属下所有从服务器，以及所有同样监视master的其他Sentinel进入主观下线的标准。

多个Sentinel设置的主观下线时长可能不同

down-after-milliseconds选项另一个需要注意的地方是，对于监视同一个主服务器的多个Sentinel来说，这些Sentinel所设置的down-after-milliseconds选项的值也可能不同，因此，当一个Sentinel将主服务器判断为主观下线时，其他Sentinel可能仍然会认为主服务器处于在线状态。举个例子，如果Sentinel1载入了以下配置：

```
sentinel monitor master 127.0.0.1 6379 2
sentinel down-after-milliseconds master 50000
```

而Sentinel2则载入了以下配置：

```
sentinel monitor master 127.0.0.1 6379 2
sentinel down-after-milliseconds master 10000
```

那么当master的断线时长超过10000毫秒之后，Sentinel2会将master判断为主观下线，而Sentinel1却认为master仍然在线。只有当master的断线时长超过50000毫秒之后，Sentinel1和Sentinel2才会都认为master进入了主观下线状态。

16.7 检查客观下线状态

当Sentinel将一个主服务器判断为主观下线之后，为了确认这个主服务器是否真的下线了，它会向同样监视这一主服务器的其他Sentinel进行询问，看它们是否也认为主服务器已经进入了下线状态（可以是主观下线或者客观下线）。当Sentinel从其他Sentinel那里接收到足够数量的已下线判断之后，Sentinel就会将从服务器判定为客观下线，并对主服务器执行故障转移操作。

16.7.1 发送SENTINEL is-master-down-by-addr命令

Sentinel使用：

```
SENTINEL is-master-down-by-addr <ip> <port> <current_epoch> <runid>
```

命令询问其他Sentinel是否同意主服务器已下线，命令中的各个参数的意义如表16-4所示。

表16-4 SENTINEL is-master-down-by-addr命令各个参数的意义

参数	意义
ip	被 Sentinel 判断为主观下线的主服务器的 IP 地址
port	被 Sentinel 判断为主观下线的主服务器的端口号
current_epoch	Sentinel 当前的配置纪元，用于选举领头 Sentinel，详细作用将在下一节说明
runid	可以是 * 符号或者 Sentinel 的运行 ID：* 符号代表命令仅仅用于检测主服务器的客观下线状态，而 Sentinel 的运行 ID 则用于选举领头 Sentinel，详细作用将在下一节说明

举个例子，如果被Sentinel判断为主观下线的主服务器的IP为127.0.0.1，端口号为6379，并且Sentinel当前的配置纪元为0，那么Sentinel将向其他Sentinel发送以下命令：

```
SENTINEL is-master-down-by-addr 127.0.0.1 6379 0 *
```

16.7.2 接收SENTINEL is-master-down-by-addr命令

当一个Sentinel（目标Sentinel）接收到另一个Sentinel（源Sentinel）发来的SENTINEL is-master-down-by命令时，目标Sentinel会分析并取出命令请求中包含的各个参数，并根据其中的主服务器IP和端口号，检查主服务器是否已下线，然后向源Sentinel返回一条包含三个参数的Multi Bulk回复作为SENTINEL is-master-down-by命令的回复：

```
1) <down_state>
2) <leader_runid>
3) <leader_epoch>
```

表16-5分别记录了这三个参数的意义。

表16-5　SENTINEL is-master-down-by-addr回复的意义

参　数	意　义
down_state	返回目标 Sentinel 对主服务器的检查结果，1 代表主服务器已下线，0 代表主服务器未下线
leader_runid	可以是 * 符号或者目标 Sentinel 的局部领头 Sentinel 的运行 ID：* 符号代表命令仅仅用于检测主服务器的下线状态，而局部领头 Sentinel 的运行 ID 则用于选举领头 Sentinel，详细作用将在下一节说明
leader_epoch	目标 Sentinel 的局部领头 Sentinel 的配置纪元，用于选举领头 Sentinel，详细作用将在下一节说明。仅在 leader_runid 的值不为 * 时有效，如果 leader_runid 的值为 *，那么 leader_epoch 总为 0

举个例子，如果一个Sentinel返回以下回复作为SENTINEL is-master-down-by-addr命令的回复：

```
1) 1
2) *
3) 0
```

那么说明Sentinel也同意主服务器已下线。

16.7.3　接收SENTINEL is-master-down-by-addr命令的回复

根据其他Sentinel发回的SENTINEL is-master-down-by-addr命令回复，Sentinel将统计其他Sentinel同意主服务器已下线的数量，当这一数量达到配置指定的判断客观下线所需的数量时，Sentinel会将主服务器实例结构flags属性的SRI_O_DOWN标识打开，表示主服务器已经进入客观下线状态，如图16-19所示。

sentinelRedisInstance
flags SRI_MASTER | SRI_S_DOWN | SRI_O_DOWN
name "master"
...

图16-19　主服务器被标记为客观下线

客观下线状态的判断条件

当认为主服务器已经进入下线状态的Sentinel的数量，超过Sentinel配置中设置的quorum参数的值，那么该Sentinel就会认为主服务器已经进入客观下线状态。比如说，如果Sentinel在启动时载入了以下配置：

```
sentinel monitor master 127.0.0.1 6379 2
```

那么包括当前Sentinel在内，只要总共有两个Sentinel认为主服务器已经进入下线状态，那么当前Sentinel就将主服务器判断为客观下线。又比如说，如果Sentinel在启动时载入了以下配置：

```
sentinel monitor master 127.0.0.1 6379 5
```

那么包括当前Sentinel在内，总共要有五个Sentinel都认为主服务器已经下线，当前Sentinel才会将主服务器判断为客观下线。

不同Sentinel判断客观下线的条件可能不同

对于监视同一个主服务器的多个Sentinel来说，它们将主服务器标判断为客观下线的条件可能也不同：当一个Sentinel将主服务器判断为客观下线时，其他Sentinel可能并不是那么认为的。比如说，对于监视同一个主服务器的五个Sentinel来说，如果Sentinel1在启动时载入了以下配置：

```
sentinel monitor master 127.0.0.1 6379 2
```

那么当五个Sentinel中有两个Sentinel认为主服务器已经下线时，Sentinel1就会将主服务器标判断为客观下线。

而对于载入了以下配置的Sentinel2来说：

```
sentinel monitor master 127.0.0.1 6379 5
```

仅有两个Sentinel认为主服务器已下线，并不会令Sentinel2将主服务器判断为客观下线。

16.8　选举领头Sentinel

当一个主服务器被判断为客观下线时，监视这个下线主服务器的各个Sentinel会进行协商，选举出一个领头Sentinel，并由领头Sentinel对下线主服务器执行故障转移操作。

以下是Redis选举领头Sentinel的规则和方法：

·所有在线的Sentinel都有被选为领头Sentinel的资格，换句话说，监视同一个主服务器的多个在线Sentinel中的任意一个都有可能成为领头Sentinel。

·每次进行领头Sentinel选举之后，不论选举是否成功，所有Sentinel的配置纪元（configuration　epoch）的值都会自增一次。配置纪元实际上就是一个计数器，并没有什么特别的。

·在一个配置纪元里面，所有Sentinel都有一次将某个Sentinel设置为局部领头Sentinel的机会，并且局部领头一旦设置，在这个配置纪元里面就不能再更改。

·每个发现主服务器进入客观下线的Sentinel都会要求其他Sentinel将自己设置为局部领头Sentinel。

·当一个Sentinel（源Sentinel）向另一个Sentinel（目标Sentinel）发送SENTINEL　is-master-down-by-addr命令，并且命令中的runid参数不是*符号而是源Sentinel的运行ID时，这表示源Sentinel要求目标Sentinel将前者设置为后者的局部领头Sentinel。

·Sentinel设置局部领头Sentinel的规则是先到先得：最先向目标Sentinel发送设置要求的源Sentinel将成为目标Sentinel的局部领头Sentinel，而之后接收到的所有设置要求都会被目标Sentinel拒绝。

·目标Sentinel在接收到SENTINEL　is-master-down-by-addr命令之后，将向源Sentinel返回一条命令回复，回复中的leader_runid参数和leader_epoch参数分别记录了目标Sentinel的局部领头Sentinel的运行ID和配置纪元。

·源Sentinel在接收到目标Sentinel返回的命令回复之后，会检查回复中leader_epoch参数的值和自己的配置纪元是否相同，如果相同的话，那么源Sentinel继续取出回复中的leader_runid参数，如果leader_runid参数的值和源Sentinel的运行ID一致，那么表示目标Sentinel将源Sentinel设置成了局部领头Sentinel。

·如果有某个Sentinel被半数以上的Sentinel设置成了局部领头Sentinel，那么这个Sentinel成为领头Sentinel。举个例子，在一个由10

个Sentinel组成的Sentinel系统里面，只要有大于等于10/2+1=6个Sentinel将某个Sentinel设置为局部领头Sentinel，那么被设置的那个Sentinel就会成为领头Sentinel。

·因为领头Sentinel的产生需要半数以上Sentinel的支持，并且每个Sentinel在每个配置纪元里面只能设置一次局部领头Sentinel，所以在一个配置纪元里面，只会出现一个领头Sentinel。

·如果在给定时限内，没有一个Sentinel被选举为领头Sentinel，那么各个Sentinel将在一段时间之后再次进行选举，直到选出领头Sentinel为止。

为了熟悉以上规则，让我们来看一个选举领头Sentinel的过程。

假设现在有三个Sentinel正在监视同一个主服务器，并且这三个Sentinel之前已经通过SENTINEL is-master-down-by-addr命令确认主服务器进入了客观下线状态，如图16-20所示。

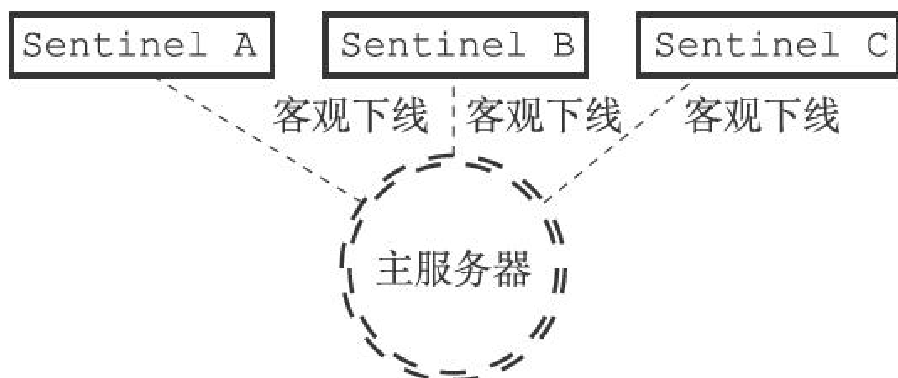


图16-20　三个Sentinel都发现主服务器已经进入了客观下线状态

那么为了选出领头Sentinel，三个Sentinel将再次向其他Sentinel发送SENTINEL is-master-down-by-addr命令，如图16-21所示。

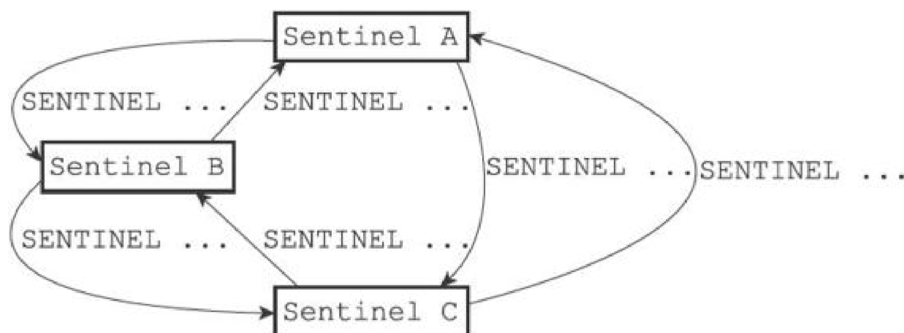


图16-21　Sentinel再次向其他Sentinel发送命令

和检测客观下线状态时发送的SENTINEL is-master-down-by-addr命令不同，Sentinel这次发送的命令会带有Sentinel自己的运行ID，例如：

```
SENTINEL is-master-down-by-
addr 127.0.0.1 6379 0 e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa
```

如果接收到这个命令的Sentinel还没有设置局部领头Sentinel的话，它就会将运行ID为e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa的Sentinel设置为自己的局部领头Sentinel，并返回类似以下的命令回复：

```
1) 1
2) e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa
3) 0
```

然后接收到命令回复的Sentinel就可以根据这一回复，统计出有多少个Sentinel将自己设置成了局部领头Sentinel。

根据命令请求发送的先后顺序不同，可能会有某个Sentinel的SENTINEL is-master-down-by-addr命令比起其他Sentinel发送的相同命令都更快到达，并最终胜出领头Sentinel的选举，然后这个领头Sentinel就可以开始对主服务器执行故障转移操作了。

16.9 故障转移

在选举产生出领头Sentinel之后，领头Sentinel将对已下线的主服务器执行故障转移操作，该操作包含以下三个步骤：

1）在已下线主服务器属下的所有从服务器里面，挑选出一个从服务器，并将其转换为主服务器。

2）让已下线主服务器属下的所有从服务器改为复制新的主服务器。

3）将已下线主服务器设置为新的主服务器的从服务器，当这个旧的主服务器重新上线时，它就会成为新的主服务器的从服务器。

16.9.1 选出新的主服务器

故障转移操作第一步要做的就是在已下线主服务器属下的所有从服务器中，挑选出一个状态良好、数据完整的从服务器，然后向这个从服务器发送SLAVEOF no one命令，将这个从服务器转换为主服务器。

新的主服务器是怎样挑选出来的

领头Sentinel会将已下线主服务器的所有从服务器保存到一个列表里面，然后按照以下规则，一项一项地对列表进行过滤：

1）删除列表中所有处于下线或者断线状态的从服务器，这可以保证列表中剩余的从服务器都是正常在线的。

2）删除列表中所有最近五秒内没有回复过领头Sentinel的INFO命令的从服务器，这可以保证列表中剩余的从服务器都是最近成功进行过通信的。

3）删除所有与已下线主服务器连接断开超过down-after-milliseconds*10毫秒的从服务器：down-after-milliseconds选项指定了判断主服务器下线所需的时间，而删除断开时长超过down-after-milliseconds*10毫秒的从服务器，则可以保证列表中剩余的从服务器都没有过早地与主服务器断开连接，换句话说，列表中剩余的从服务器保存的数据都是比较新的。

之后，领头Sentinel将根据从服务器的优先级，对列表中剩余的从服务器进行排序，并选出其中优先级最高的从服务器。

如果有多个具有相同最高优先级的从服务器，那么领头

Sentinel将按照从服务器的复制偏移量，对具有相同最高优先级的所有从服务器进行排序，并选出其中偏移量最大的从服务器（复制偏移量最大的从服务器就是保存着最新数据的从服务器）。

最后，如果有多个优先级最高、复制偏移量最大的从服务器，那么领头Sentinel将按照运行ID对这些从服务器进行排序，并选出其中运行ID最小的从服务器。

图16-22展示了在一次故障转移操作中，领头Sentinel向被选中的从服务器server2发送SLAVEOF no one命令的情形。

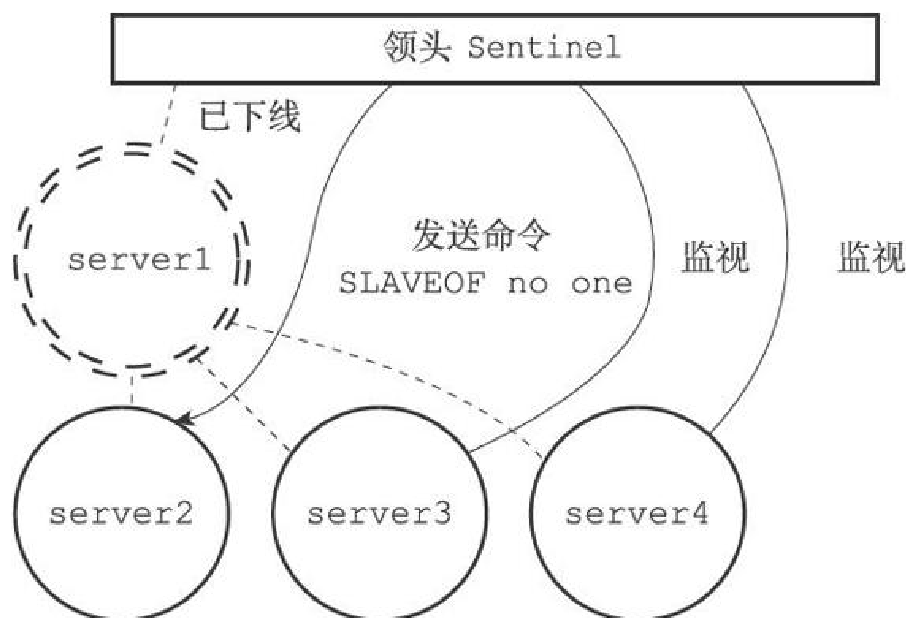


图16-22　将server2升级为主服务器

在发送SLAVEOF no one命令之后，领头Sentinel会以每秒一次的频率（平时是每十秒一次），向被升级的从服务器发送INFO命令，并观察命令回复中的角色（role）信息，当被升级服务器的role从原来的slave变为master时，领头Sentinel就知道被选中的从服务器已经顺利升级为主服务器了。

例如，在图16-22展示的例子中，领头Sentinel会一直向server2发送INFO命令，当server2返回的命令回复从：

```
# Replication
role:slave
...
# Other sections
...
```

变为：

```
# Replication
role:master
...
# Other sections
...
```

的时候，领头Sentinel就知道server2已经成功升级为主服务器了。

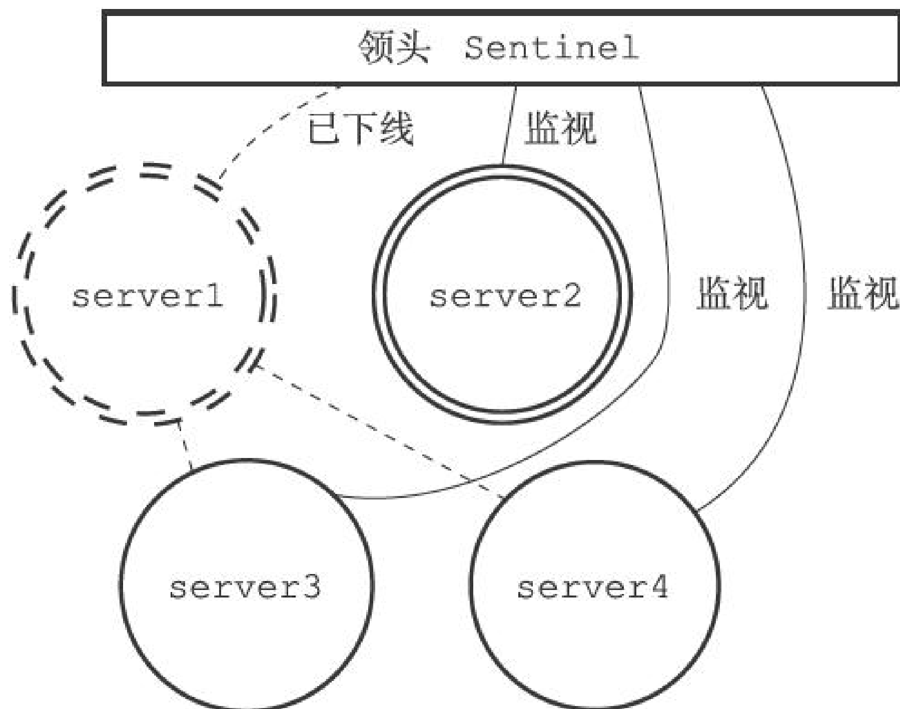


图16-23　server2成功升级为主服务器

图16-23展示了server2升级成功之后，各个服务器和领头Sentinel的样子。

16.9.2　修改从服务器的复制目标

当新的主服务器出现之后，领头Sentinel下一步要做的就是，让已下线主服务器属下的所有从服务器去复制新的主服务器，这一动作可以通过向从服务器发送SLAVEOF命令来实现。

图16-24展示了在故障转移操作中，领头Sentinel向已下线主服务器server1的两个从服务器server3和server4发送SLAVEOF命令，让它们复制新的主服务器server2的例子。

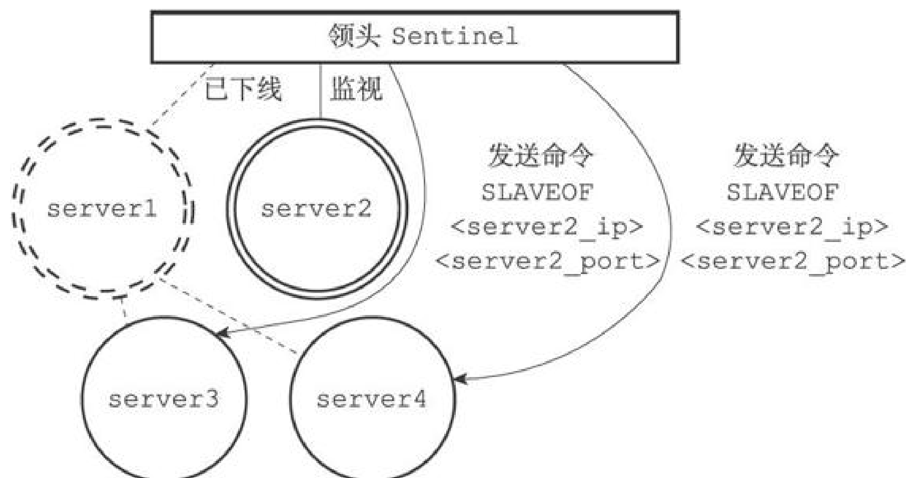


图16-24　让从服务器复制新的主服务器

图16-25展示了server3和server4成为server2的从服务器之后，各个服务器以及领头Sentinel的样子。

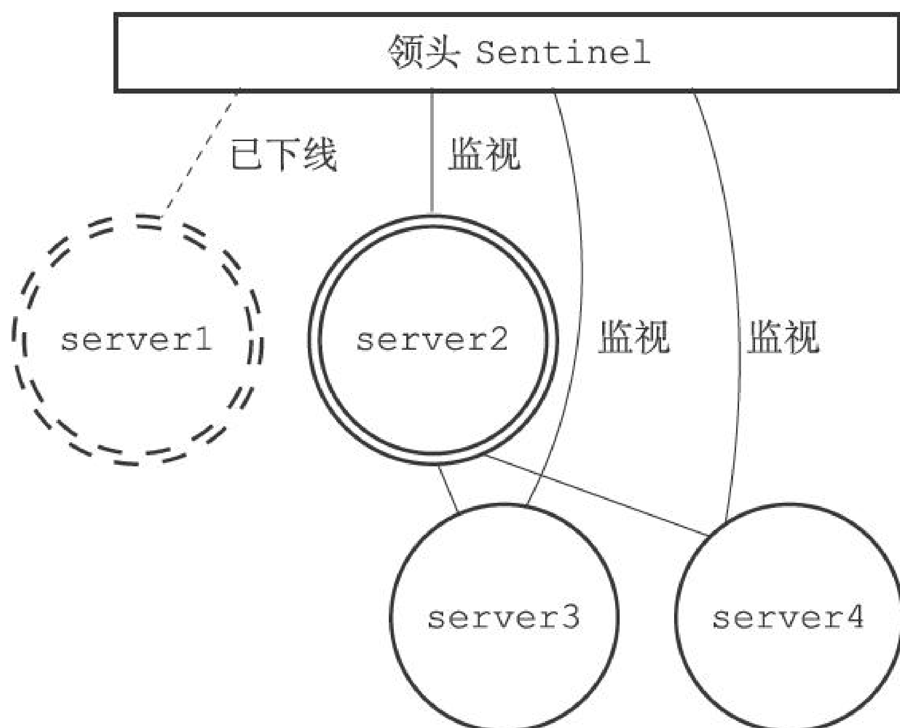


图16-25　server3和server4成为server2的从服务器

16.9.3　将旧的主服务器变为从服务器

故障转移操作最后要做的是，将已下线的主服务器设置为新的主服

务器的从服务器。比如说，图16-26就展示了被领头Sentinel设置为从服务器之后，服务器server1的样子。

因为旧的主服务器已经下线，所以这种设置是保存在server1对应的实例结构里面的，当server1重新上线时，Sentinel就会向它发送SLAVEOF命令，让它成为server2的从服务器。

例如，图16-27就展示了server1重新上线并成为server2的从服务器的例子。

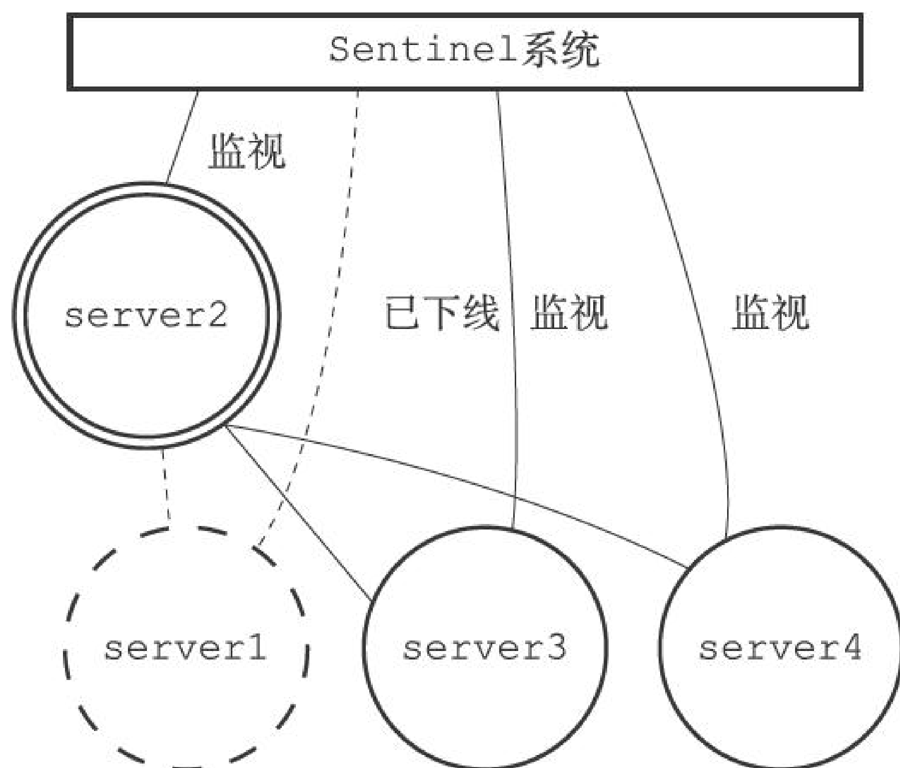


图16-26　server1被设置为新主服务器的从服务器

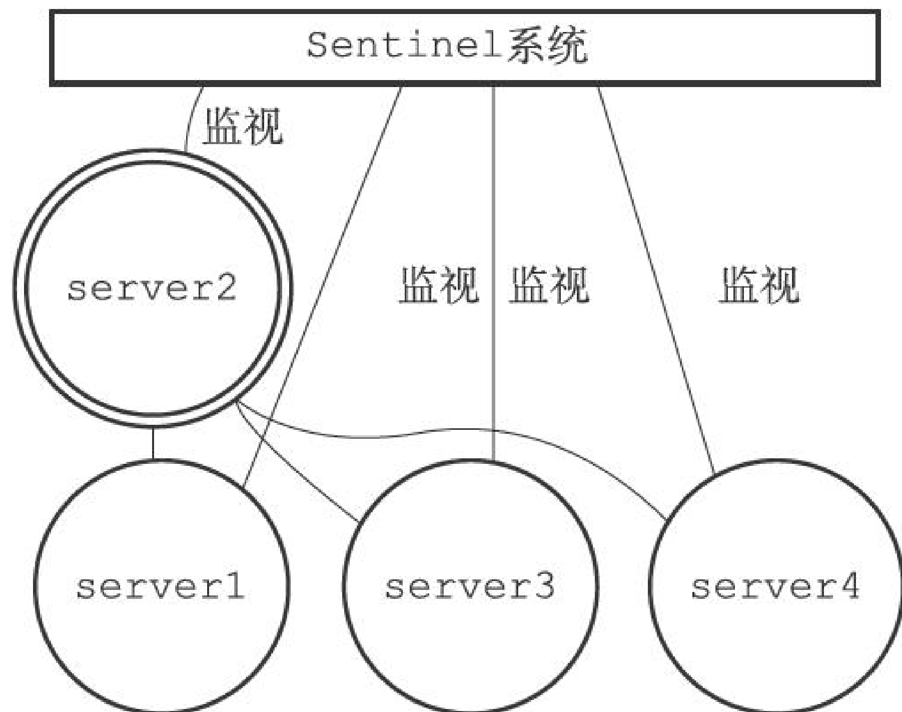


图16-27　server1重新上线并成为server2的从服务器

16.10 重点回顾

·Sentinel只是一个运行在特殊模式下的Redis服务器，它使用了和普通模式不同的命令表，所以Sentinel模式能够使用的命令和普通Redis服务器能够使用的命令不同。

·Sentinel会读入用户指定的配置文件，为每个要被监视的主服务器创建相应的实例结构，并创建连向主服务器的命令连接和订阅连接，其中命令连接用于向主服务器发送命令请求，而订阅连接则用于接收指定频道的消息。

·Sentinel通过向主服务器发送INFO命令来获得主服务器属下所有从服务器的地址信息，并为这些从服务器创建相应的实例结构，以及连向这些从服务器的命令连接和订阅连接。

·在一般情况下，Sentinel以每十秒一次的频率向被监视的主服务器和从服务器发送INFO命令，当主服务器处于下线状态，或者Sentinel正在对主服务器进行故障转移操作时，Sentinel向从服务器发送INFO命令的频率会改为每秒一次。

·对于监视同一个主服务器和从服务器的多个Sentinel来说，它们会以每两秒一次的频率，通过向被监视服务器的_sentinel_:hello频道发送消息来向其他Sentinel宣告自己的存在。

·每个Sentinel也会从_sentinel_:hello频道中接收其他Sentinel发来的信息，并根据这些信息为其他Sentinel创建相应的实例结构，以及命令连接。

·Sentinel只会与主服务器和从服务器创建命令连接和订阅连接，Sentinel与Sentinel之间则只创建命令连接。

·Sentinel以每秒一次的频率向实例（包括主服务器、从服务器、其他Sentinel）发送PING命令，并根据实例对PING命令的回复来判断实例是否在线，当一个实例在指定的时长中连续向Sentinel发送无效回复时，Sentinel会将这个实例判断为主观下线。

·当Sentinel将一个主服务器判断为主观下线时，它会向同样监视这个主服务器的其他Sentinel进行询问，看它们是否同意这个主服务器已经进入主观下线状态。

·当Sentinel收集到足够多的主观下线投票之后，它会将主服务器判断为客观下线，并发起一次针对主服务器的故障转移操作。

16.11 参考资料

Sentinel系统选举领头Sentinel的方法是对Raft算法的领头选举方法的实现，关于这一方法的详细信息可以观看Raft算法的作者录制的“Raft教程”视频：http://v.youku.com/v_show/id_XNjQxOTk5MTk2.html，或者Raft算法的论文。

第17章　集群

Redis集群是Redis提供的分布式数据库方案，集群通过分片（sharding）来进行数据共享，并提供复制和故障转移功能。

本节将对集群的节点、槽指派、命令执行、重新分片、转向、故障转移、消息等各个方面进行介绍。

17.1 节点

一个Redis集群通常由多个节点（node）组成，在刚开始的时候，每个节点都是相互独立的，它们都处于一个只包含自己的集群当中，要组建一个真正可工作的集群，我们必须将各个独立的节点连接起来，构成一个包含多个节点的集群。

连接各个节点的工作可以使用CLUSTER MEET命令来完成，该命令的格式如下：

```
CLUSTER MEET <ip> <port>
```

向一个节点node发送CLUSTER MEET命令，可以让node节点与ip和port所指定的节点进行握手（handshake），当握手成功时，node节点就会将ip和port所指定的节点添加到node节点当前所在的集群中。

举个例子，假设现在有三个独立的节点127.0.0.1:7000、127.0.0.1:7001、127.0.0.1:7002（下文省略IP地址，直接使用端口号来区分各个节点），我们首先使用客户端连上节点7000，通过发送CLUSTER NODE命令可以看到，集群目前只包含7000自己一个节点：

```
$ redis-cli -c -p 7000
127.0.0.1:7000> CLUSTER NODES
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0 connected
```

通过向节点7000发送以下命令，我们可以将节点7001添加到节点7000所在的集群里面：

```
127.0.0.1:7000> CLUSTER MEET 127.0.0.1 7001
OK
127.0.0.1:7000> CLUSTER NODES
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master -
 0 1388204746210 0 connected
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0 connected
```

继续向节点7000发送以下命令，我们可以将节点7002也添加到节点7000和节点7001所在的集群里面：

```
127.0.0.1:7000> CLUSTER MEET 127.0.0.1 7002
OK
127.0.0.1:7000> CLUSTER NODES
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master -
 0 1388204848376 0 connected
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master -
 0 1388204847977 0 connected
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0 connected
```

现在，这个集群里面包含了7000、7001和7002三个节点，图17-1至17-5展示了这三个节点进行握手的整个过程。



图17-1　三个独立的节点

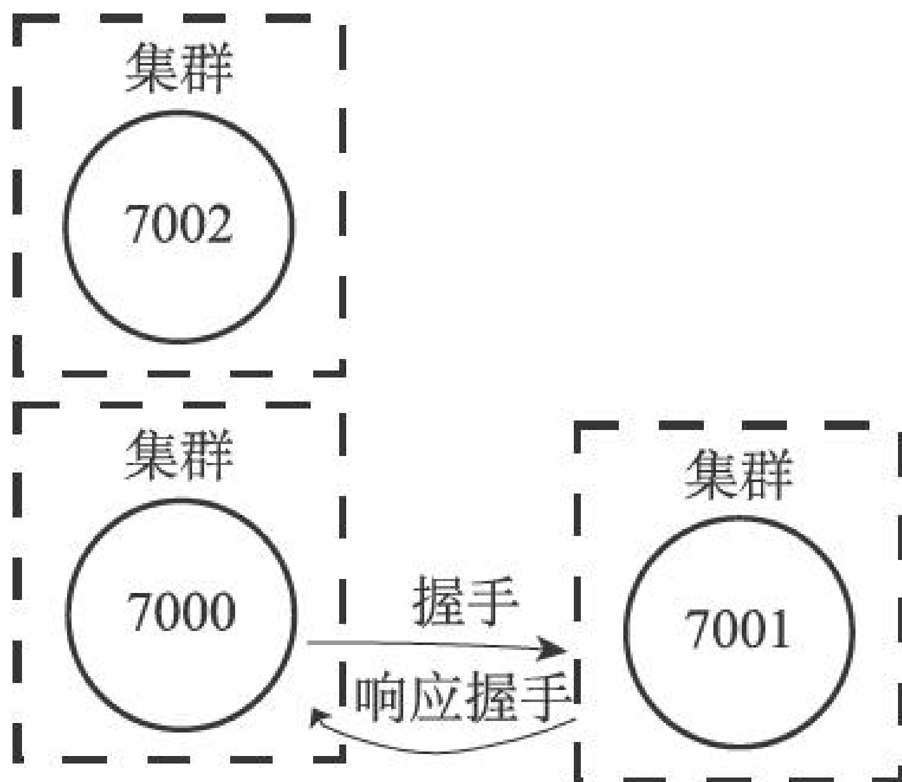


图17-2　节点7000和7001进行握手

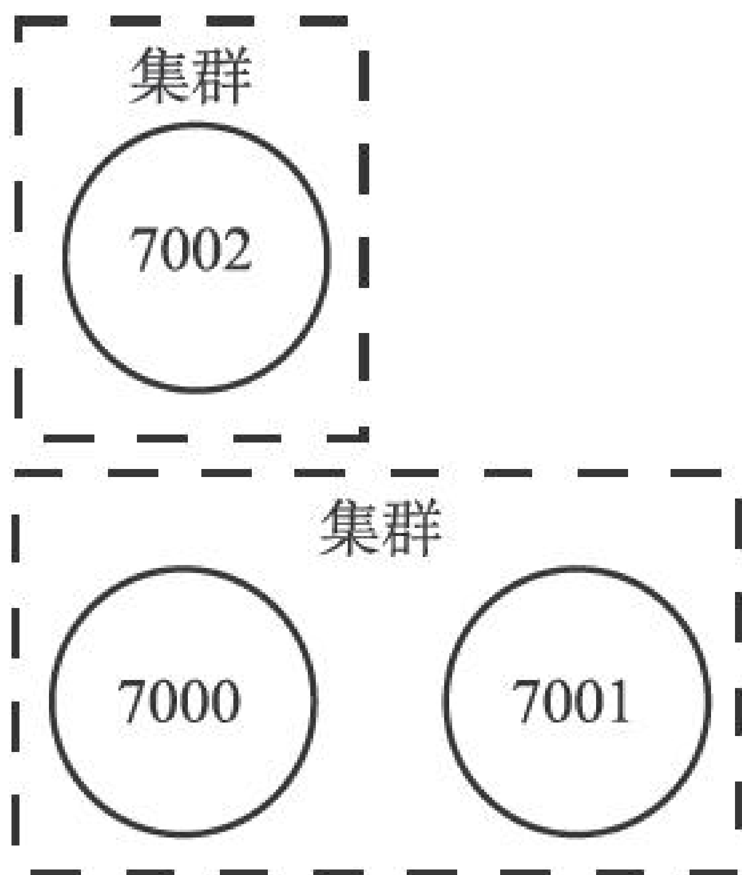


图17-3　握手成功的7000与7001处于同一个集群

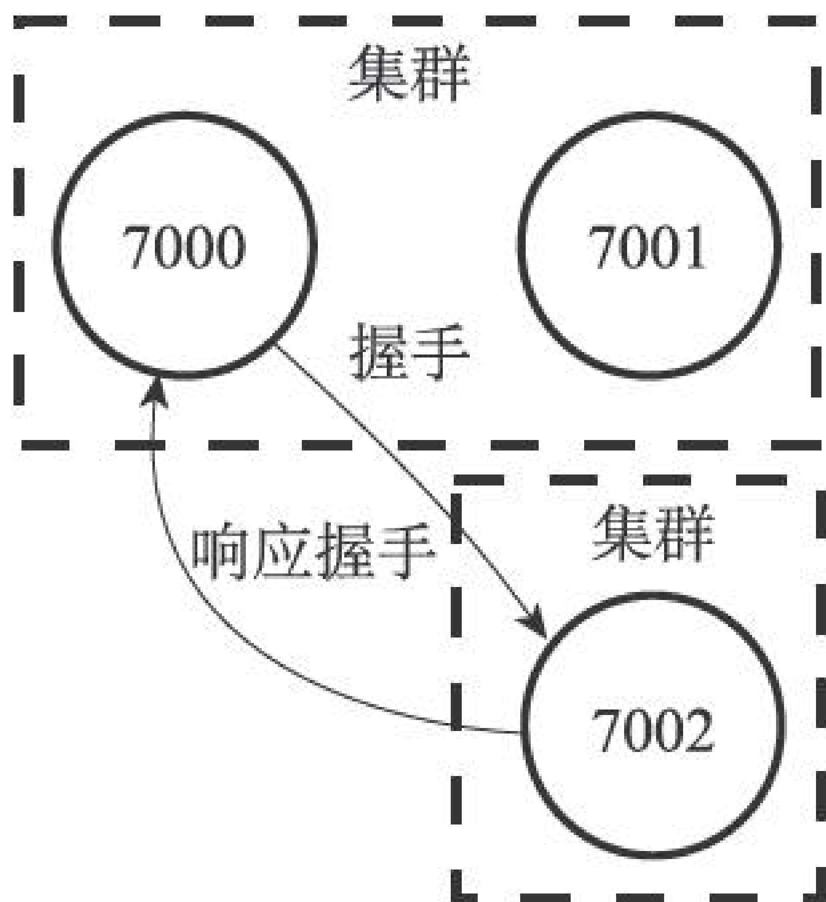


图17-4　节点7000与节点7002进行握手

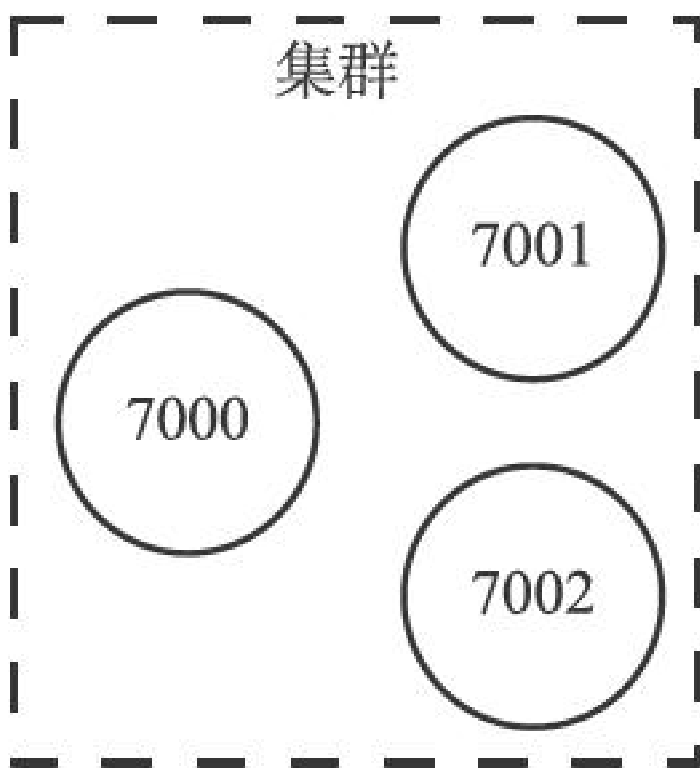


图17-5　握手成功的三个节点处于同一个集群

本节接下来的内容将介绍启动节点的方法、与集群有关的数据结构，以及CLUSTER MEET命令的实现原理。

17.1.1　启动节点

一个节点就是一个运行在集群模式下的Redis服务器，Redis服务器在启动时会根据cluster-enabled配置选项是否为yes来决定是否开启服务器的集群模式，如图17-6所示。

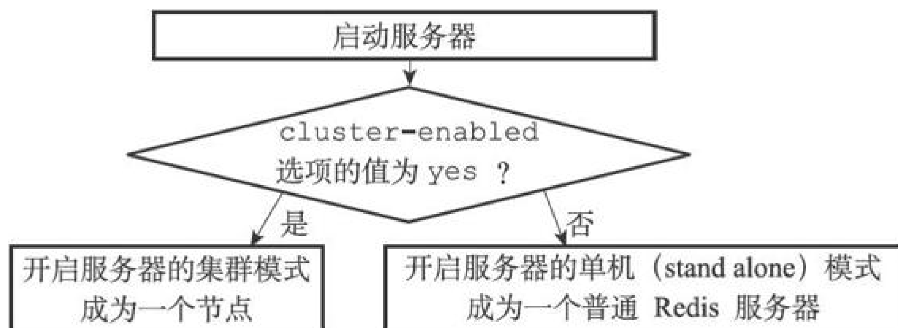


图17-6　服务器判断是否开启集群模式的过程

节点（运行在集群模式下的Redis服务器）会继续使用所有在单机模

式中使用的服务器组件，比如说：

·节点会继续使用文件事件处理器来处理命令请求和返回命令回复。

·节点会继续使用时间事件处理器来执行serverCron函数，而serverCron函数又会调用集群模式特有的clusterCron函数。clusterCron函数负责执行在集群模式下需要执行的常规操作，例如向集群中的其他节点发送Gossip消息，检查节点是否断线，或者检查是否需要对下线节点进行自动故障转移等。

·节点会继续使用数据库来保存键值对数据，键值对依然会是各种不同类型的对象。

·节点会继续使用RDB持久化模块和AOF持久化模块来执行持久化工作。

·节点会继续使用发布与订阅模块来执行PUBLISH、SUBSCRIBE等命令。

·节点会继续使用复制模块来进行节点的复制工作。

·节点会继续使用Lua脚本环境来执行客户端输入的Lua脚本。

除此之外，节点会继续使用redisServer结构来保存服务器的状态，使用redisClient结构来保存客户端的状态，至于那些只有在集群模式下才会用到的数据，节点将它们保存到了cluster.h/clusterNode结构、cluster.h/clusterLink结构，以及cluster.h/clusterState结构里面，接下来的一节将对这三种数据结构进行介绍。

17.1.2　集群数据结构

clusterNode结构保存了一个节点的当前状态，比如节点的创建时间、节点的名字、节点当前的配置纪元、节点的IP地址和端口号等等。

每个节点都会使用一个clusterNode结构来记录自己的状态，并为集群中的所有其他节点（包括主节点和从节点）都创建一个相应的clusterNode结构，以此来记录其他节点的状态：

```
struct clusterNode {
  //
创建节点的时间
  mstime_t ctime;
  //
节点的名字，由40
个十六进制字符组成
  //
例如68eef66df23420a5862208ef5b1a7005b806f2ff
  char name[REDIS_CLUSTER_NAMELEN];
  //
节点标识
```

```
    //
使用各种不同的标识值记录节点的角色（比如主节点或者从节点），
    //
以及节点目前所处的状态（比如在线或者下线）。
    int flags;
    //
节点当前的配置纪元，用于实现故障转移
    uint64_t configEpoch;
    //
节点的IP
地址
    char ip[REDIS_IP_STR_LEN];
    //
节点的端口号
    int port;
    //
保存连接节点所需的有关信息
    clusterLink *link;
    // ...
};
```

clusterNode结构的link属性是一个clusterLink结构，该结构保存了连接节点所需的有关信息，比如套接字描述符，输入缓冲区和输出缓冲区：

```
typedef struct clusterLink {
    //
连接的创建时间
    mstime_t ctime;
    // TCP
套接字描述符
    int fd;
    //
输出缓冲区，保存着等待发送给其他节点的消息（message
）。
    sds sndbuf;
    //
输入缓冲区，保存着从其他节点接收到的消息。
    sds rcvbuf;
    //
与这个连接相关联的节点，如果没有的话就为NULL
    struct clusterNode *node;
} clusterLink;
```

redisClient结构和clusterLink结构的相同和不同之处

redisClient结构和clusterLink结构都有自己的套接字描述符和输入、输出缓冲区，这两个结构的区别在于，redisClient结构中的套接字和缓冲区是用于连接客户端的，而clusterLink结构中的套接字和缓冲区则是用于连接节点的。

最后，每个节点都保存着一个clusterState结构，这个结构记录了在当前节点的视角下，集群目前所处的状态，例如集群是在线还是下线，集群包含多少个节点，集群当前的配置纪元，诸如此类：

```
typedef struct clusterState {
  //
指向当前节点的指针
  clusterNode *myself;
  //
集群当前的配置纪元，用于实现故障转移
  uint64_t currentEpoch;
  //
集群当前的状态：是在线还是下线
  int state;
  //
集群中至少处理着一个槽的节点的数量
  int size;
  //
集群节点名单（包括myself
节点）
  //
字典的键为节点的名字，字典的值为节点对应的clusterNode
结构
  dict *nodes;
  // ...
} clusterState;
```

以前面介绍的7000、7001、7002三个节点为例，图17-7展示了节点7000创建的clusterState结构，这个结构从节点7000的角度记录了集群以及集群包含的三个节点的当前状态（为了空间考虑，图中省略了clusterNode结构的一部分属性）：

·结构的currentEpoch属性的值为0，表示集群当前的配置纪元为0。

·结构的size属性的值为0，表示集群目前没有任何节点在处理槽，因此结构的state属性的值为REDIS_CLUSTER_FAIL，这表示集群目前处于下线状态。

·结构的nodes字典记录了集群目前包含的三个节点，这三个节点分别由三个clusterNode结构表示，其中myself指针指向代表节点7000的clusterNode结构，而字典中的另外两个指针则分别指向代表节点7001和代表节点7002的clusterNode结构，这两个节点是节点7000已知的在集群中的其他节点。

·三个节点的clusterNode结构的flags属性都是REDIS_NODE_MASTER，说明三个节点都是主节点。

节点7001和节点7002也会创建类似的clusterState结构：

·不过在节点7001创建的clusterState结构中，myself指针将指向代表节点7001的clusterNode结构，而节点7000和节点7002则是集群中的其他节点。

·而在节点7002创建的clusterState结构中，myself指针将指向代表节点7002的clusterNode结构，而节点7000和节点7001则是集群中的其他节点。

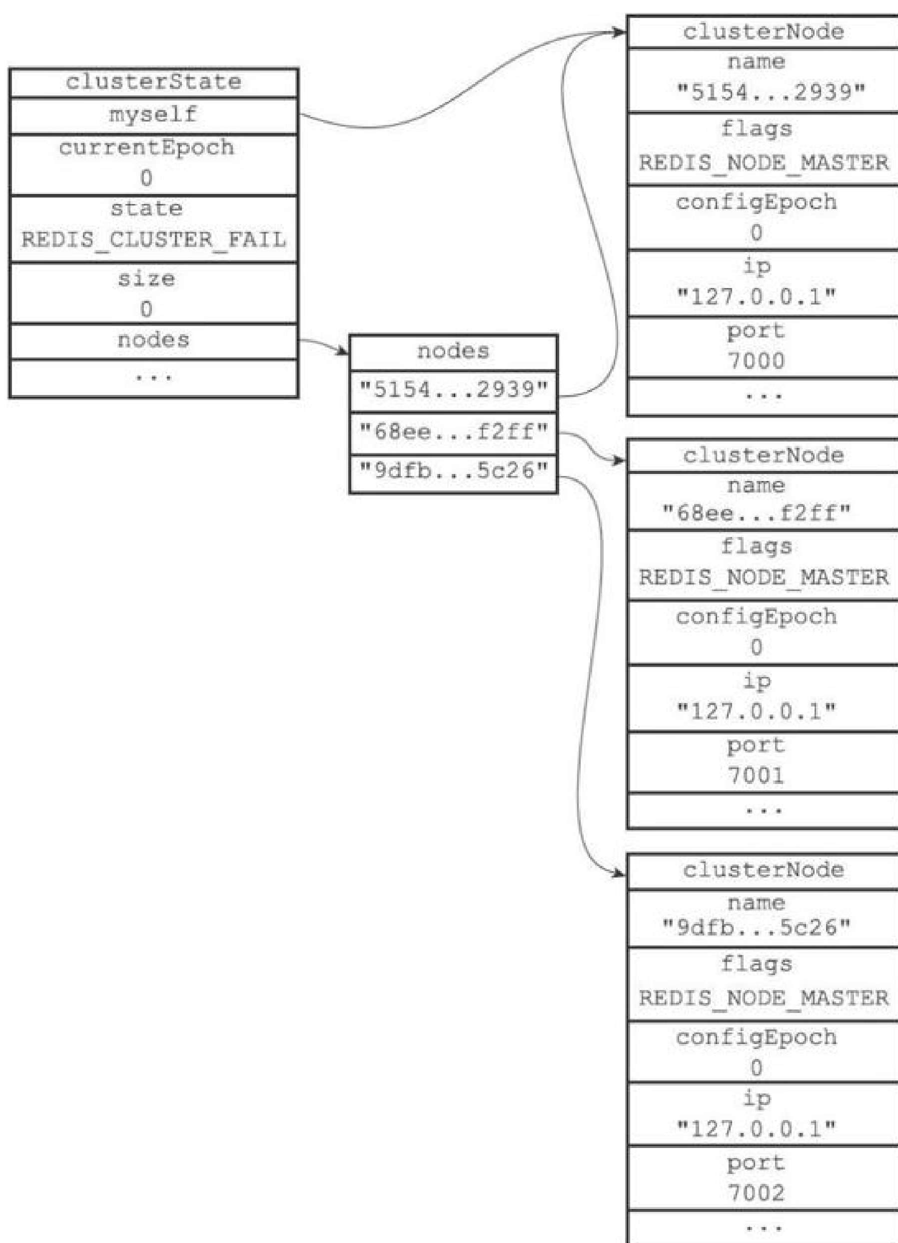


图17-7　节点7000创建的clusterState结构

17.1.3　CLUSTER MEET命令的实现

通过向节点A发送CLUSTER　MEET命令，客户端可以让接收命令的节点A将另一个节点B添加到节点A当前所在的集群里面：

```
CLUSTER MEET <ip> <port>
```

收到命令的节点A将与节点B进行握手（handshake），以此来确认彼此的存在，并为将来的进一步通信打好基础：

1）节点A会为节点B创建一个clusterNode结构，并将该结构添加到自己的clusterState.nodes字典里面。

2）之后，节点A将根据CLUSTER　MEET命令给定的IP地址和端口号，向节点B发送一条MEET消息（message）。

3）如果一切顺利，节点B将接收到节点A发送的MEET消息，节点B会为节点A创建一个clusterNode结构，并将该结构添加到自己的clusterState.nodes字典里面。

4）之后，节点B将向节点A返回一条PONG消息。

5）如果一切顺利，节点A将接收到节点B返回的PONG消息，通过这条PONG消息节点A可以知道节点B已经成功地接收到了自己发送的MEET消息。

6）之后，节点A将向节点B返回一条PING消息。

7）如果一切顺利，节点B将接收到节点A返回的PING消息，通过这条PING消息节点B可以知道节点A已经成功地接收到了自己返回的PONG消息，握手完成。

图17-8展示了以上步骤描述的握手过程。

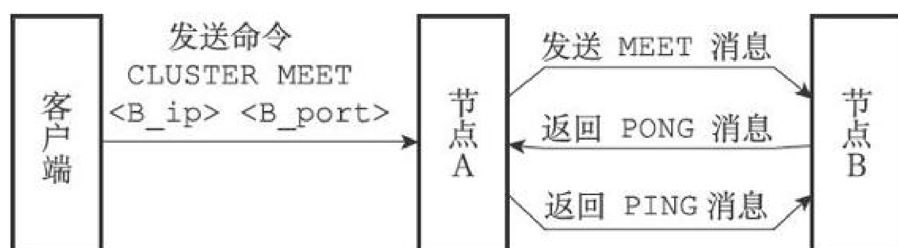


图17-8　节点的握手过程

之后，节点A会将节点B的信息通过Gossip协议传播给集群中的其他节点，让其他节点也与节点B进行握手，最终，经过一段时间之后，节点B会被集群中的所有节点认识。

17.2　槽指派

Redis集群通过分片的方式来保存数据库中的键值对：集群的整个数据库被分为16384个槽（slot），数据库中的每个键都属于这16384个槽的其中一个，集群中的每个节点可以处理0个或最多16384个槽。

当数据库中的16384个槽都有节点在处理时，集群处于上线状态（ok）；相反地，如果数据库中有任何一个槽没有得到处理，那么集群处于下线状态（fail）。

在上一节，我们使用CLUSTER　MEET命令将7000、7001、7002三个节点连接到了同一个集群里面，不过这个集群目前仍然处于下线状态，因为集群中的三个节点都没有在处理任何槽：

```
127.0.0.1:7000> CLUSTER INFO
cluster_state:fail
cluster_slots_assigned:0
cluster_slots_ok:0
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:3
cluster_size:0
cluster_current_epoch:0
cluster_stats_messages_sent:110
cluster_stats_messages_received:28
```

通过向节点发送CLUSTER　ADDSLOTS命令，我们可以将一个或多个槽指派（assign）给节点负责：

```
CLUSTER ADDSLOTS <slot> [slot ...]
```

举个例子，执行以下命令可以将槽0至槽5000指派给节点7000负责：

```
127.0.0.1:7000> CLUSTER ADDSLOTS 0 1 2 3 4 ... 5000
OK
127.0.0.1:7000> CLUSTER NODES
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master -
 0 1388316664849 0 connected
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master -
 0 1388316665850 0 connected
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master -
 0 0 0 connected 0-5000
```

为了让7000、7001、7002三个节点所在的集群进入上线状态，我们继续执行以下命令，将槽5001至槽10000指派给节点7001负责：

```
127.0.0.1:7001> CLUSTER ADDSLOTS 5001 5002 5003 5004 ... 10000
OK
```

然后将槽10001至槽16383指派给7002负责：

```
127.0.0.1:7002> CLUSTER ADDSLOTS 10001 10002 10003 10004 ... 16383
OK
```

当以上三个CLUSTER ADDSLOTS命令都执行完毕之后，数据库中的16384个槽都已经被指派给了相应的节点，集群进入上线状态：

```
127.0.0.1:7000> CLUSTER INFO
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:3
cluster_size:3
cluster_current_epoch:0
cluster_stats_messages_sent:2699
cluster_stats_messages_received:2617
127.0.0.1:7000> CLUSTER NODES
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master -
 0 1388317426165 0 connected 10001-16383
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master -
 0 1388317427167 0 connected 5001-10000
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master -
 0 0 0 connected 0-5000
```

本节接下来的内容将首先介绍节点保存槽指派信息的方法，以及节点之间传播槽指派信息的方法，之后再介绍CLUSTER ADDSLOTS命令的实现。

17.2.1 记录节点的槽指派信息

clusterNode结构的slots属性和numslot属性记录了节点负责处理哪些槽：

```
struct clusterNode {
  // ...
  unsigned char slots[16384/8];
  int numslots;
  // ...
};
```

slots属性是一个二进制位数组（bit array），这个数组的长度为16384/8＝2048个字节，共包含16384个二进制位。

Redis以0为起始索引，16383为终止索引，对slots数组中的16384个二进制位进行编号，并根据索引i上的二进制位的值来判断节点是否负责处理槽i：

·如果slots数组在索引i上的二进制位的值为1，那么表示节点负责处理槽i。

·如果slots数组在索引i上的二进制位的值为0，那么表示节点不负责处理槽i。

图17-9展示了一个slots数组示例：这个数组索引0至索引7上的二进制位的值都为1，其余所有二进制位的值都为0，这表示节点负责处理槽0至槽7。

字节	slots[0]								slots[1] ~ slots[2047]								
索引	0	1	2	3	4	5	6	7	8	9	10	11	12	...	16381	16382	16383
值	1	1	1	1	1	1	1	1	0	0	0	0	0	...	0	0	0

图17-9　一个slots数组示例

图17-10展示了另一个slots数组示例：这个数组索引1、3、5、8、9、10上的二进制位的值都为1，而其余所有二进制位的值都为0，这表示节点负责处理槽1、3、5、8、9、10。

字节	slots[0]								slots[1]								...	slots[2047]		
索引	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	...	16382	16383
值	0	1	0	1	0	1	0	0	1	1	1	0	0	0	0	0	...	...	0	0

图17-10　另一个slots数组示例

因为取出和设置slots数组中的任意一个二进制位的值的复杂度仅为O（1），所以对于一个给定节点的slots数组来说，程序检查节点是否负责处理某个槽，又或者将某个槽指派给节点负责，这两个动作的复杂度都是O（1）。

至于numslots属性则记录节点负责处理的槽的数量，也即是slots数组中值为1的二进制位的数量。

比如说，对于图17-9所示的slots数组来说，节点处理的槽数量为8，而对于图17-10所示的slots数组来说，节点处理的槽数量为6。

17.2.2　传播节点的槽指派信息

一个节点除了会将自己负责处理的槽记录在clusterNode结构的slots属性和numslots属性之外，它还会将自己的slots数组通过消息发送给集群中的其他节点，以此来告知其他节点自己目前负责处理哪些槽。

举个例子，对于前面展示的包含7000、7001、7002三个节点的集群来说：

·节点7000会通过消息向节点7001和节点7002发送自己的slots数组，以此来告知这两个节点，自己负责处理槽0至槽5000，如图17-11所示。

·节点7001会通过消息向节点7000和节点7002发送自己的slots数组，以此来告知这两个节点，自己负责处理槽5001至槽10000，如图17-12所示。

·节点7002会通过消息向节点7000和节点7001发送自己的slots数组，以此来告知这两个节点，自己负责处理槽10001至槽16383，如图17-13所示。

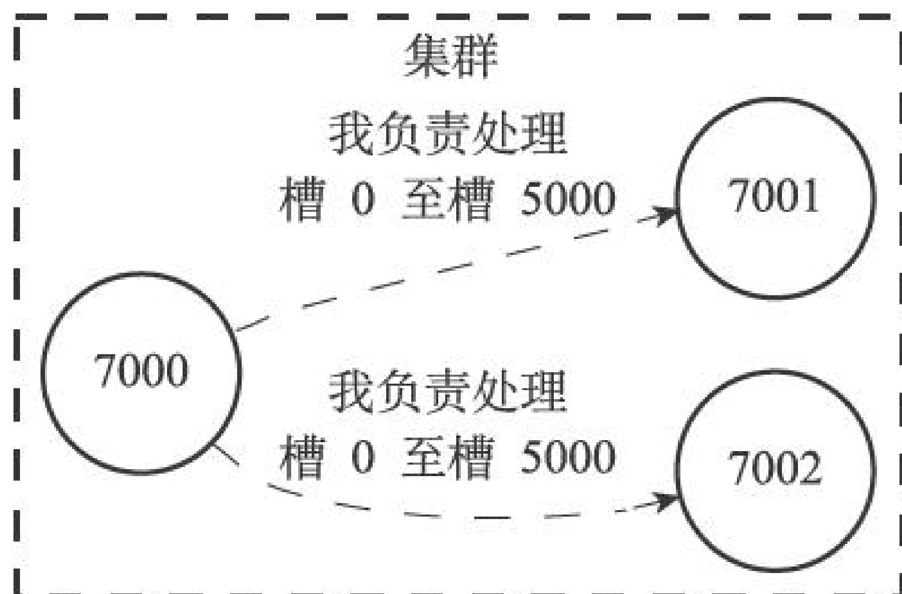


图17-11　7000告知7001和7002自己负责处理的槽

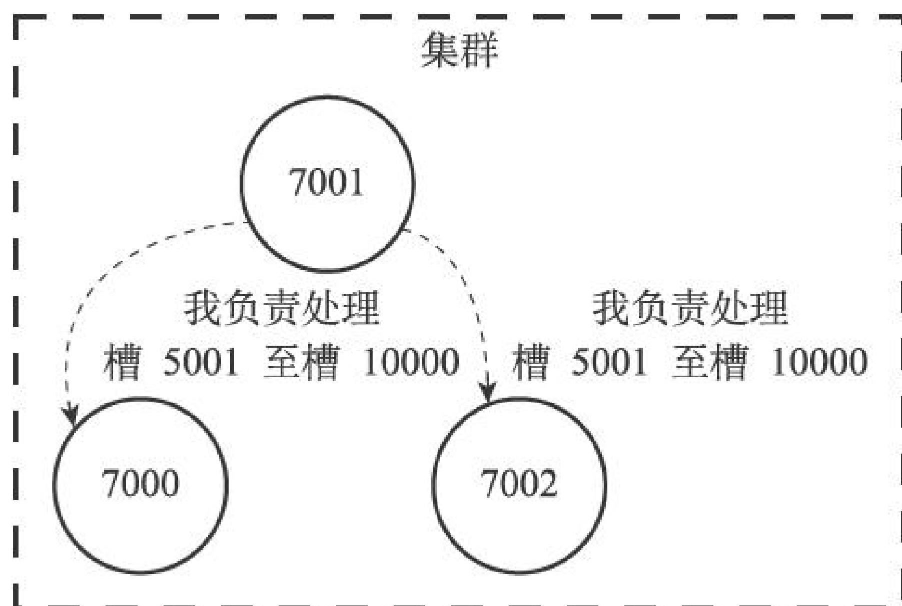


图17-12　7001告知7000和7002自己负责处理的槽

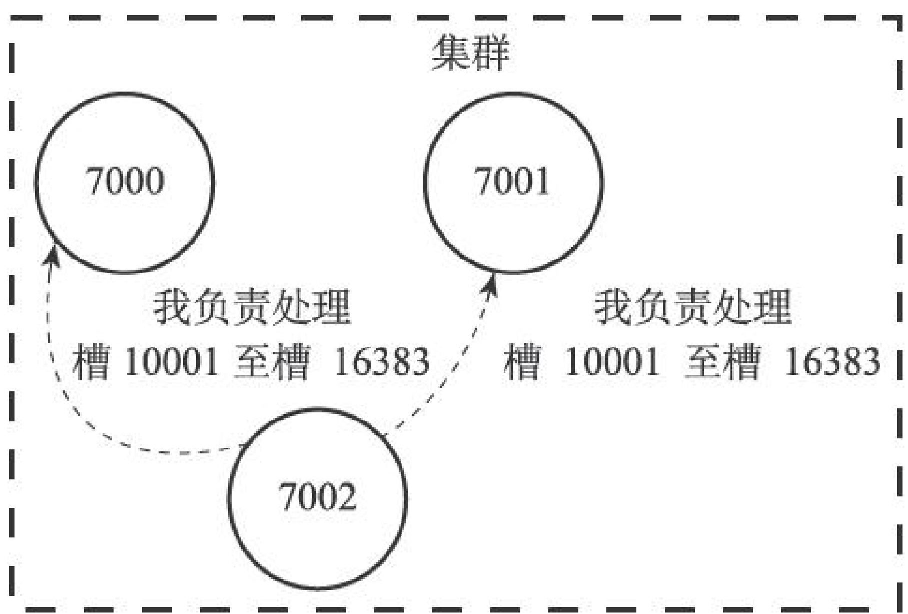


图17-13　7002告知7000和7001自己负责处理的槽

当节点A通过消息从节点B那里接收到节点B的slots数组时，节点A会在自己的clusterState.nodes字典中查找节点B对应的clusterNode结构，并对结构中的slots数组进行保存或者更新。

因为集群中的每个节点都会将自己的slots数组通过消息发送给集群中的其他节点，并且每个接收到slots数组的节点都会将数组保存到相应节点的clusterNode结构里面，因此，集群中的每个节点都会知道数据库中的16384个槽分别被指派给了集群中的哪些节点。

17.2.3　记录集群所有槽的指派信息

clusterState结构中的slots数组记录了集群中所有16384个槽的指派信息：

```
typedef struct clusterState {
  // ...
  clusterNode *slots[16384];
  // ...
} clusterState;
```

slots数组包含16384个项，每个数组项都是一个指向clusterNode结构的指针：

·如果slots[i]指针指向NULL，那么表示槽i尚未指派给任何节点。

·如果slots[i]指针指向一个clusterNode结构，那么表示槽i已经指派给了clusterNode结构所代表的节点。

举个例子，对于7000、7001、7002三个节点来说，它们的clusterState结构的slots数组将会是图17-14所示的样子：

·数组项slots[0]至slots[5000]的指针都指向代表节点7000的clusterNode结构，表示槽0至5000都指派给了节点7000。

·数组项slots[5001]至slots[10000]的指针都指向代表节点7001的clusterNode结构，表示槽5001至10000都指派给了节点7001。

·数组项slots[10001]至slots[16383]的指针都指向代表节点7002的clusterNode结构，表示槽10001至16383都指派给了节点7002。

如果只将槽指派信息保存在各个节点的clusterNode.slots数组里，会出现一些无法高效地解决的问题，而clusterState.slots数组的存在解决了这些问题：

·如果节点只使用clusterNode.slots数组来记录槽的指派信息，那么为了知道槽i是否已经被指派，或者槽i被指派给了哪个节点，程序需要遍历clusterState.nodes字典中的所有clusterNode结构，检查这些结构的slots数组，直到找到负责处理槽i的节点为止，这个过程的复杂度为O（N），其中N为clusterState.nodes字典保存的clusterNode结构的数量。

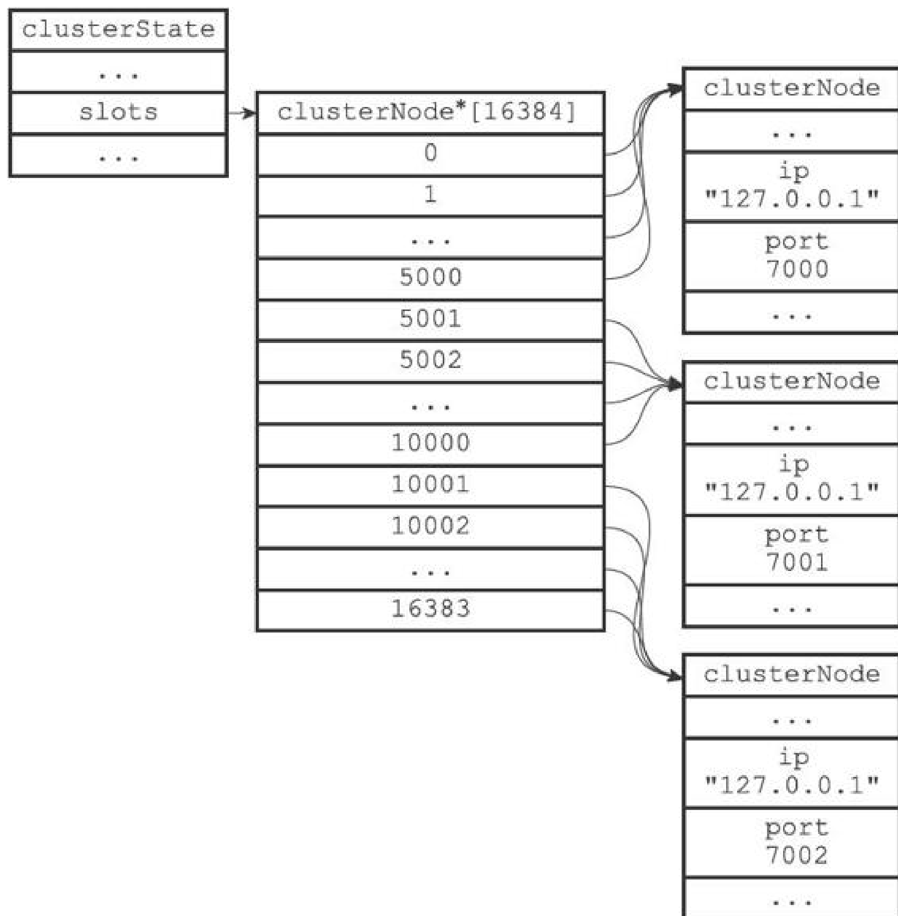


图17-14　clusterState结构的slots数组

·而通过将所有槽的指派信息保存在clusterState.slots数组里面，程序要检查槽i是否已经被指派，又或者取得负责处理槽i的节点，只需要访问clusterState.slots[i]的值即可，这个操作的复杂度仅为O（1）。

举个例子，对于图17-14所示的slots数组来说，如果程序需要知道槽10002被指派给了哪个节点，那么只要访问数组项slots[10002]，就可以马上知道槽10002被指派给了节点7002，如图17-15所示。

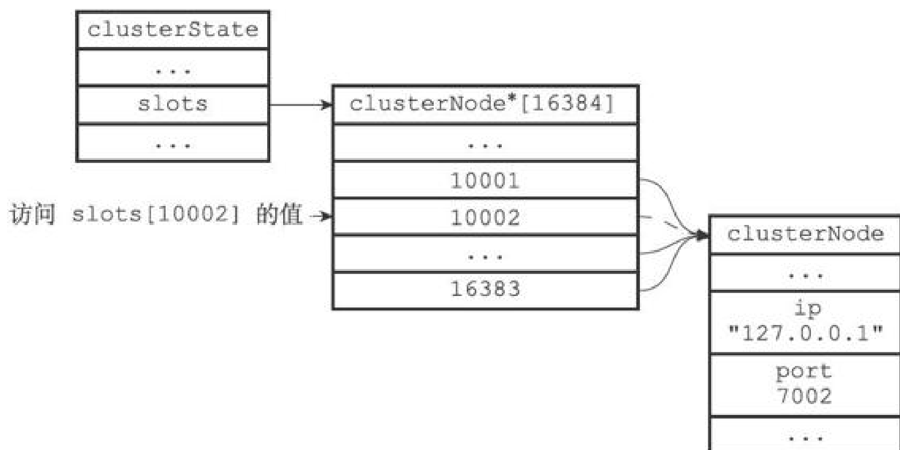


图17-15　访问slots[10002]的值

要说明的一点是，虽然clusterState.slots数组记录了集群中所有槽的指派信息，但使用clusterNode结构的slots数组来记录单个节点的槽指派信息仍然是有必要的：

·因为当程序需要将某个节点的槽指派信息通过消息发送给其他节点时，程序只需要将相应节点的clusterNode.slots数组整个发送出去就可以了。

·另一方面，如果Redis不使用clusterNode.slots数组，而单独使用clusterState.slots数组的话，那么每次要将节点A的槽指派信息传播给其他节点时，程序必须先遍历整个clusterState.slots数组，记录节点A负责处理哪些槽，然后才能发送节点A的槽指派信息，这比直接发送clusterNode.slots数组要麻烦和低效得多。

clusterState.slots数组记录了集群中所有槽的指派信息，而clusterNode.slots数组只记录了clusterNode结构所代表的节点的槽指派信息，这是两个slots数组的关键区别所在。

17.2.4　CLUSTER ADDSLOTS命令的实现

CLUSTER　ADDSLOTS命令接受一个或多个槽作为参数，并将所有输入的槽指派给接收该命令的节点负责：

```
CLUSTER ADDSLOTS <slot> [slot ...]
```

CLUSTER ADDSLOTS命令的实现可以用以下伪代码来表示：

```
def CLUSTER_ADDSLOTS(*all_input_slots):
    #
遍历所有输入槽，检查它们是否都是未指派槽
    for i in all_input_slots:
```

```
        #
如果有哪怕一个槽已经被指派给了某个节点
        #
那么向客户端返回错误，并终止命令执行
        if clusterState.slots[i] != NULL:
            reply_error()
            return
    #
如果所有输入槽都是未指派槽
    #
那么再次遍历所有输入槽，将这些槽指派给当前节点
    for i in all_input_slots:
        #
设置clusterState
结构的slots
数组
        #
将slots[i]
的指针指向代表当前节点的clusterNode
结构
        clusterState.slots[i] = clusterState.myself
        #
访问代表当前节点的clusterNode
结构的slots
数组
        #
将数组在索引i
上的二进制位设置为1
        setSlotBit(clusterState.myself.slots, i)
```

举个例子，图17-16展示了一个节点的clusterState结构，clusterState.slots数组中的所有指针都指向NULL，并且clusterNode.slots数组中的所有二进制位的值都是0，这说明当前节点没有被指派任何槽，并且集群中的所有槽都是未指派的。

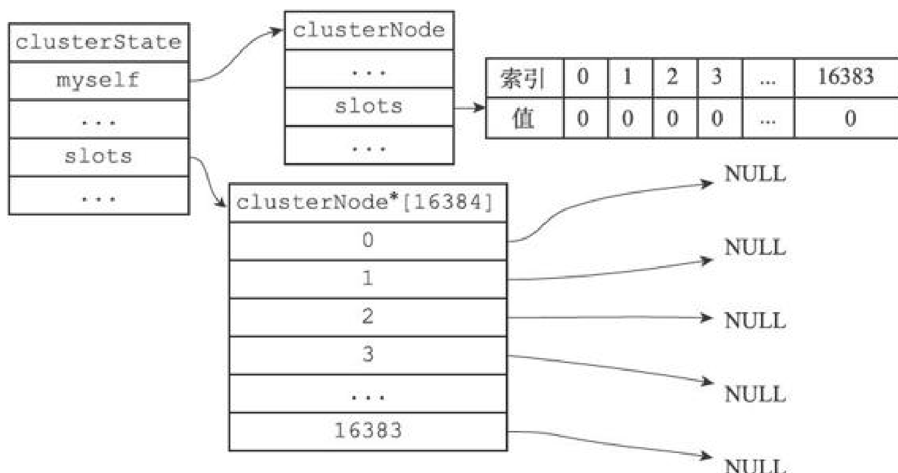


图17-16　节点的clusterState结构

当客户端对17-16所示的节点执行命令：

```
CLUSTER ADDSLOTS 1 2
```

将槽1和槽2指派给节点之后，节点的clusterState结构将被更新成图

17-17所示的样子：

·clusterState.slots数组在索引1和索引2上的指针指向了代表当前节点的clusterNode结构。

·并且clusterNode.slots数组在索引1和索引2上的位被设置成了1。

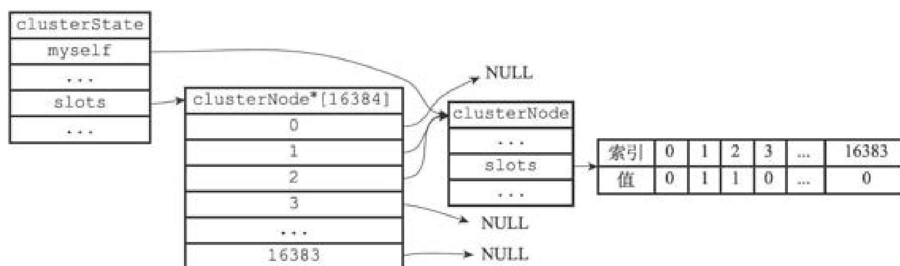


图17-17　执行CLUSTER ADDSLOTS命令之后的clusterState结构

最后，在CLUSTER　ADDSLOTS命令执行完毕之后，节点会通过发送消息告知集群中的其他节点，自己目前正在负责处理哪些槽。

17.3 在集群中执行命令

在对数据库中的16384个槽都进行了指派之后，集群就会进入上线状态，这时客户端就可以向集群中的节点发送数据命令了。

当客户端向节点发送与数据库键有关的命令时，接收命令的节点会计算出命令要处理的数据库键属于哪个槽，并检查这个槽是否指派给了自己：

·如果键所在的槽正好就指派给了当前节点，那么节点直接执行这个命令。

·如果键所在的槽并没有指派给当前节点，那么节点会向客户端返回一个MOVED错误，指引客户端转向（redirect）至正确的节点，并再次发送之前想要执行的命令。

图17-18展示了这两种情况的判断流程。

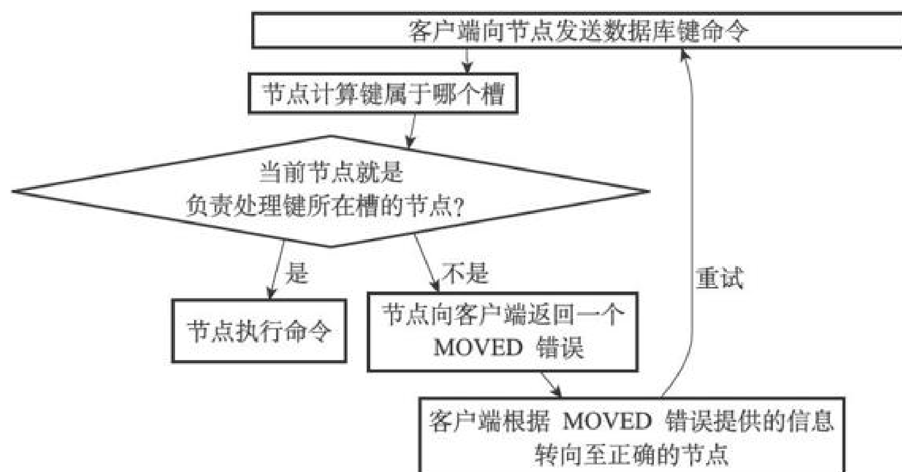


图17-18　判断客户端是否需要转向的流程

举个例子，如果我们在之前提到的，由7000、7001、7002三个节点组成的集群中，用客户端连上节点7000，并发送以下命令，那么命令会直接被节点7000执行：

```
127.0.0.1:7000> SET date "2013-12-31"
OK
```

因为键date所在的槽2022正是由节点7000负责处理的。

但是，如果我们执行以下命令，那么客户端会先被转向至节点7001，然后再执行命令：

```
127.0.0.1:7000> SET msg "happy new year!"
-> Redirected to slot [6257] located at 127.0.0.1:7001
OK
127.0.0.1:7001> GET msg
"happy new year!"
```

这是因为键msg所在的槽6257是由节点7001负责处理的，而不是由最初接收命令的节点7000负责处理：

·当客户端第一次向节点7000发送SET命令的时候，节点7000会向客户端返回MOVED错误，指引客户端转向至节点7001。

·当客户端转向到节点7001之后，客户端重新向节点7001发送SET命令，这个命令会被节点7001成功执行。

本节接下来的内容将介绍计算键所属槽的方法，节点判断某个槽是否由自己负责的方法，以及MOVED错误的实现方法，最后，本节还会介绍节点和单机Redis服务器保存键值对数据的相同和不同之处。

17.3.1　计算键属于哪个槽

节点使用以下算法来计算给定键key属于哪个槽：

```
def slot_number(key):
    return CRC16(key) & 16383
```

其中CRC16（key）语句用于计算键key的CRC-16校验和，而&16383语句则用于计算出一个介于0至16383之间的整数作为键key的槽号。

使用CLUSTER KEYSLOT＜key＞命令可以查看一个给定键属于哪个槽：

```
127.0.0.1:7000> CLUSTER KEYSLOT "date"
(integer) 2022
127.0.0.1:7000> CLUSTER KEYSLOT "msg"
(integer) 6257
127.0.0.1:7000> CLUSTER KEYSLOT "name"
(integer) 5798
127.0.0.1:7000> CLUSTER KEYSLOT "fruits"
(integer) 14943
```

CLUSTER　KEYSLOT命令就是通过调用上面给出的槽分配算法来实现的，以下是该命令的伪代码实现：

```
def CLUSTER_KEYSLOT(key):
    #
计算槽号
    slot = slot_number(key)
    #
将槽号返回给客户端
    reply_client(slot)
```

17.3.2 判断槽是否由当前节点负责处理

当节点计算出键所属的槽i之后，节点就会检查自己在clusterState.slots数组中的项i，判断键所在的槽是否由自己负责：

1）如果clusterState.slots[i]等于clusterState.myself，那么说明槽i由当前节点负责，节点可以执行客户端发送的命令。

2）如果clusterState.slots[i]不等于clusterState.myself，那么说明槽i并非由当前节点负责，节点会根据clusterState.slots[i]指向的clusterNode结构所记录的节点IP和端口号，向客户端返回MOVED错误，指引客户端转向至正在处理槽i的节点。

举个例子，假设图17-19为节点7000的clusterState结构：

·当客户端向节点7000发送命令SET date"2013-12-31"的时候，节点首先计算出键date属于槽2022，然后检查得出clusterState.slots[2022]等于clusterState.myself，这说明槽2022正是由节点7000负责，于是节点7000直接执行这个SET命令，并将结果返回给发送命令的客户端。

·当客户端向节点7000发送命令SET msg"happy new year！"的时候，节点首先计算出键msg属于槽6257，然后检查clusterState.slots[6257]是否等于clusterState.myself，结果发现两者并不相等：这说明槽6257并非由节点7000负责处理，于是节点7000访问clusterState.slots[6257]所指向的clusterNode结构，并根据结构中记录的IP地址127.0.0.1和端口号7001，向客户端返回错误MOVED 6257 127.0.0.1:7001，指引节点转向至正在负责处理槽6257的节点7001。

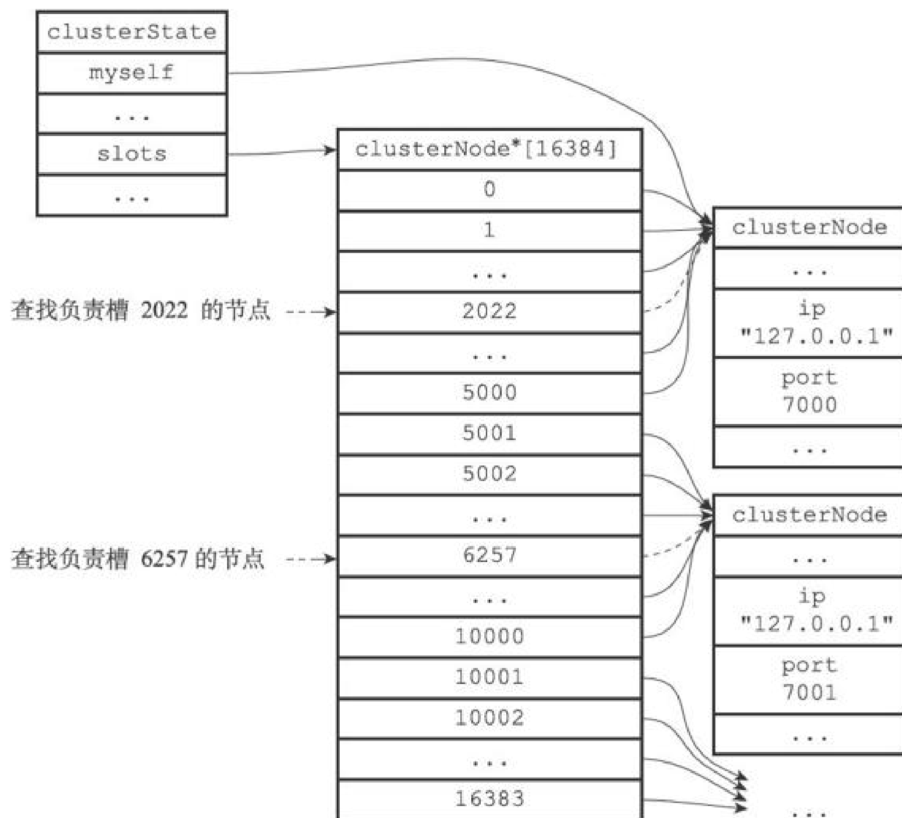


图17-19　节点7000的clusterState结构

17.3.3　MOVED错误

当节点发现键所在的槽并非由自己负责处理的时候，节点就会向客户端返回一个MOVED错误，指引客户端转向至正在负责槽的节点。

MOVED错误的格式为：

```
MOVED <slot> <ip>:<port>
```

其中slot为键所在的槽，而ip和port则是负责处理槽slot的节点的IP地址和端口号。例如错误：

```
MOVED 10086 127.0.0.1:7002
```

表示槽10086正由IP地址为127.0.0.1，端口号为7002的节点负责。

又例如错误：

```
MOVED 789 127.0.0.1:7000
```

表示槽789正由IP地址为127.0.0.1，端口号为7000的节点负责。

当客户端接收到节点返回的MOVED错误时，客户端会根据MOVED错误中提供的IP地址和端口号，转向至负责处理槽slot的节点，并向该节点重新发送之前想要执行的命令。以前面的客户端从节点7000转向至7001的情况作为例子：

```
127.0.0.1:7000> SET msg "happy new year!"
-> Redirected to slot [6257] located at 127.0.0.1:7001
OK
127.0.0.1:7001>
```

图17-20展示了客户端向节点7000发送SET命令，并获得MOVED错误的过程。

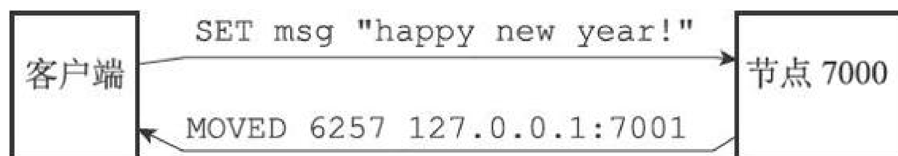


图17-20　节点7000向客户端返回MOVED错误

而图17-21则展示了客户端根据MOVED错误，转向至节点7001，并重新发送SET命令的过程。



图17-21　客户端根据MOVED错误的指示转向至节点7001

一个集群客户端通常会与集群中的多个节点创建套接字连接，而所谓的节点转向实际上就是换一个套接字来发送命令。

如果客户端尚未与想要转向的节点创建套接字连接，那么客户端会先根据MOVED错误提供的IP地址和端口号来连接节点，然后再进行转向。

被隐藏的MOVED错误

集群模式的redis-cli客户端在接收到MOVED错误时，并不会打印出MOVED错误，而是根据MOVED错误自动进行节点转向，并打印出转向信息，所以我们是看不见节点返回的MOVED错误的：

```
$ redis-cli -c -p 7000 # 
集群模式
127.0.0.1:7000> SET msg "happy new year!"
-> Redirected to slot [6257] located at 127.0.0.1:7001
OK
127.0.0.1:7001>
```

但是，如果我们使用单机（stand alone）模式的redis-cli客户端，再次向节点7000发送相同的命令，那么MOVED错误就会被客户端打印出来：

```
$ redis-cli -p 7000 # 
单机模式
127.0.0.1:7000> SET msg "happy new year!"
(error) MOVED 6257 127.0.0.1:7001
127.0.0.1:7000>
```

这是因为单机模式的redis-cli客户端不清楚MOVED错误的作用，所以它只会直接将MOVED错误直接打印出来，而不会进行自动转向。

17.3.4　节点数据库的实现

集群节点保存键值对以及键值对过期时间的方式，与第9章里面介绍的单机Redis服务器保存键值对以及键值对过期时间的方式完全相同。

节点和单机服务器在数据库方面的一个区别是，节点只能使用0号数据库，而单机Redis服务器则没有这一限制。

举个例子，图17-22展示了节点7000的数据库状态，数据库中包含列表键"lst"，哈希键"book"，以及字符串键"date"，其中键"lst"和键"book"带有过期时间。

另外，除了将键值对保存在数据库里面之外，节点还会用clusterState结构中的slots_to_keys跳跃表来保存槽和键之间的关系：

```
typedef struct clusterState {
  // ...
  zskiplist *slots_to_keys;
  // ...
} clusterState;
```

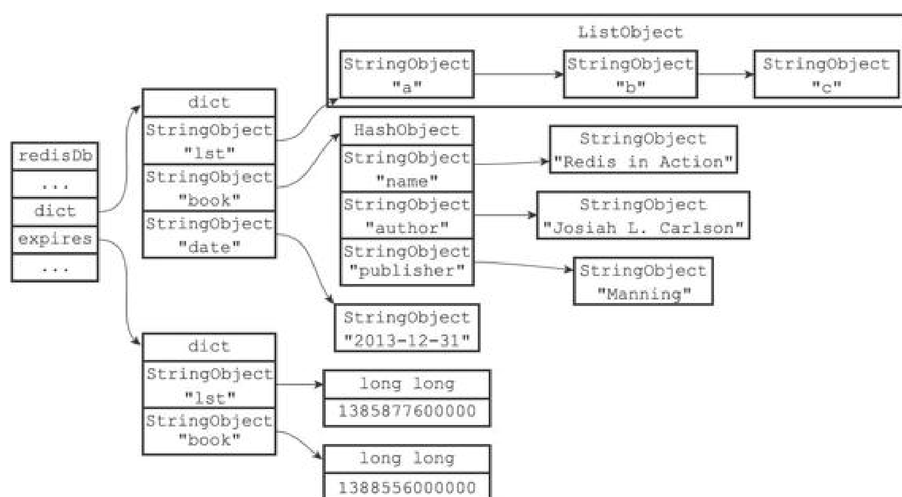


图17-22　节点7000的数据库

slots_to_keys跳跃表每个节点的分值（score）都是一个槽号，而每个节点的成员（member）都是一个数据库键：

·每当节点往数据库中添加一个新的键值对时，节点就会将这个键以及键的槽号关联到slots_to_keys跳跃表。

·当节点删除数据库中的某个键值对时，节点就会在slots_to_keys跳跃表解除被删除键与槽号的关联。

举个例子，对于图17-22所示的数据库，节点7000将创建类似图17-23所示的slots_to_keys跳跃表：

·键"book"所在跳跃表节点的分值为1337.0，这表示键"book"所在的槽为1337。

·键"date"所在跳跃表节点的分值为2022.0，这表示键"date"所在的槽为2022。

·键"lst"所在跳跃表节点的分值为3347.0，这表示键"lst"所在的槽为3347。

通过在slots_to_keys跳跃表中记录各个数据库键所属的槽，节点可以很方便地对属于某个或某些槽的所有数据库键进行批量操作，例如命令CLUSTER GETKEYSINSLOT＜slot＞＜count＞命令可以返回最多count个属于槽slot的数据库键，而这个命令就是通过遍历slots_to_keys跳跃表来实现的。

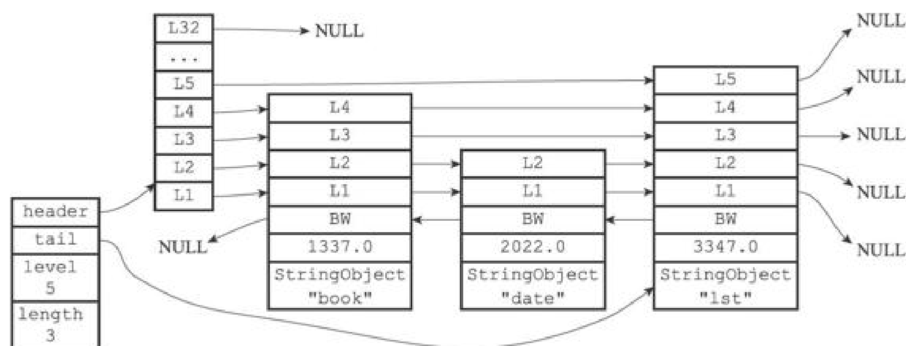


图17-23　节点7000的slots_to_keys跳跃表

17.4　重新分片

Redis集群的重新分片操作可以将任意数量已经指派给某个节点（源节点）的槽改为指派给另一个节点（目标节点），并且相关槽所属的键值对也会从源节点被移动到目标节点。

重新分片操作可以在线（online）进行，在重新分片的过程中，集群不需要下线，并且源节点和目标节点都可以继续处理命令请求。

举个例子，对于之前提到的，包含7000、7001、7002三个节点的集群来说，我们可以向这个集群添加一个IP为127.0.0.1，端口号为7003的节点（后面简称节点7003）：

```
$ redis-cli -c -p 7000
127.0.0.1:7000> CLUSTER MEET 127.0.0.1 7003
OK
127.0.0.1:7000> cluster nodes
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master -
 0 0 0 connected 0-5000
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master -
 0 1388635782831 0 connected 5001-10000
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master -
 0 1388635782831 0 connected 10001-16383
04579925484ce537d3410d7ce97bd2e260c459a2 127.0.0.1:7003 master -
 0 1388635782330 0 connected
```

然后通过重新分片操作，将原本指派给节点7002的槽15001至16383改为指派给节点7003。

以下是重新分片操作执行之后，节点的槽分配状态：

```
127.0.0.1:7000> cluster nodes
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master -0 0 0 connected 0-5000
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master -0 1388635782831 0 con
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master -0 1388635782831 0 con
04579925484ce537d3410d7ce97bd2e260c459a2 127.0.0.1:7003 master -0 1388635782330 0 con
```

重新分片的实现原理

Redis集群的重新分片操作是由Redis的集群管理软件redis-trib负责执行的，Redis提供了进行重新分片所需的所有命令，而redis-trib则通过向源节点和目标节点发送命令来进行重新分片操作。

redis-trib对集群的单个槽slot进行重新分片的步骤如下：

1）redis-trib对目标节点发送CLUSTER SETSLOT <slot> IMPORTING <source_id> 命令，让目标节点准备好从源节点导入（import）属于槽slot的键值对。

2）redis-trib对源节点发送CLUSTER

SETSLOT＜slot＞MIGRATING＜target_id＞命令，让源节点准备好将属于槽slot的键值对迁移（migrate）至目标节点。

3）redis-trib向源节点发送CLUSTER GETKEYSINSLOT＜slot＞＜count＞命令，获得最多count个属于槽slot的键值对的键名（key name）。

4）对于步骤3获得的每个键名，redis-trib都向源节点发送一个MIGRATE＜target_ip＞＜target_port＞＜key_name＞0＜timeout＞命令，将被选中的键原子地从源节点迁移至目标节点。

5）重复执行步骤3和步骤4，直到源节点保存的所有属于槽slot的键值对都被迁移至目标节点为止。每次迁移键的过程如图17-24所示。

6）redis-trib向集群中的任意一个节点发送CLUSTER SETSLOT＜slot＞NODE＜target_id＞命令，将槽slot指派给目标节点，这一指派信息会通过消息发送至整个集群，最终集群中的所有节点都会知道槽slot已经指派给了目标节点。

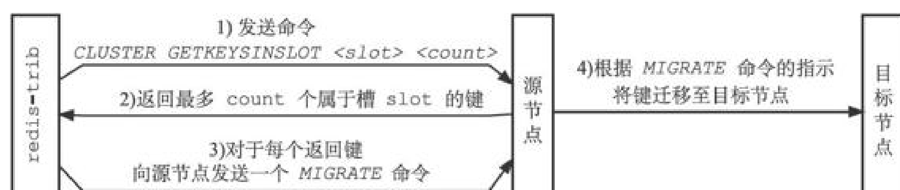


图17-24　迁移键的过程

图17-25展示了对槽slot进行重新分片的整个过程。

如果重新分片涉及多个槽，那么redis-trib将对每个给定的槽分别执行上面给出的步骤。

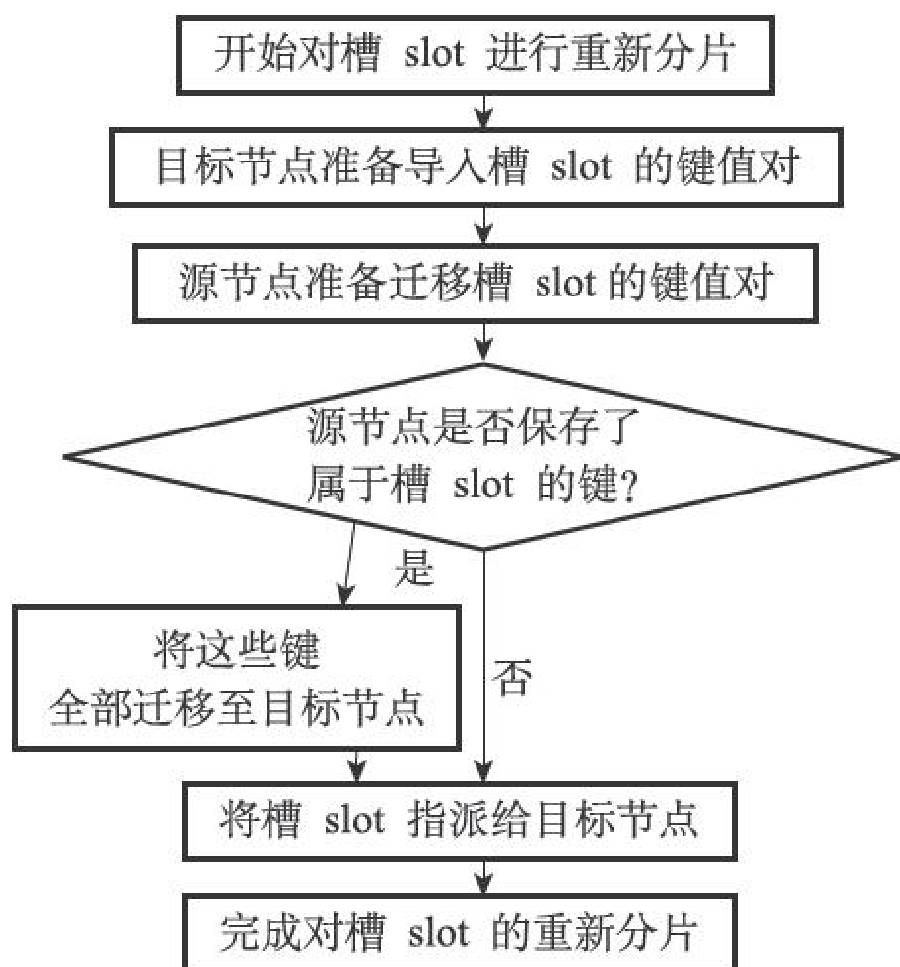


图17-25　对槽slot进行重新分片的过程

17.5 ASK错误

在进行重新分片期间，源节点向目标节点迁移一个槽的过程中，可能会出现这样一种情况：属于被迁移槽的一部分键值对保存在源节点里面，而另一部分键值对则保存在目标节点里面。

当客户端向源节点发送一个与数据库键有关的命令，并且命令要处理的数据库键恰好就属于正在被迁移的槽时：

·源节点会先在自己的数据库里面查找指定的键，如果找到的话，就直接执行客户端发送的命令。

·相反地，如果源节点没能在自己的数据库里面找到指定的键，那么这个键有可能已经被迁移到了目标节点，源节点将向客户端返回一个ASK错误，指引客户端转向正在导入槽的目标节点，并再次发送之前想要执行的命令。

图17-26展示了源节点判断是否需要向客户端发送ASK错误的整个过程。

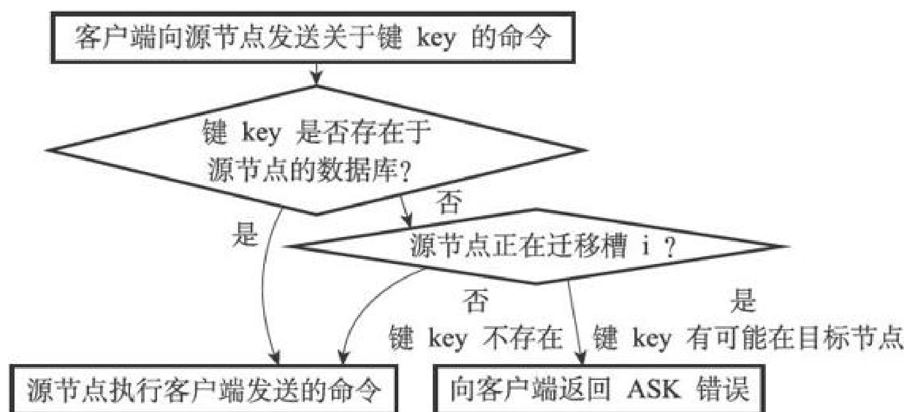


图17-26　判断是否发送ASK错误的过程

举个例子，假设节点7002正在向节点7003迁移槽16198，这个槽包含"is"和"love"两个键，其中键"is"还留在节点7002，而键"love"已经被迁移到了节点7003。

如果我们向节点7002发送关于键"is"的命令，那么这个命令会直接被节点7002执行：

```
127.0.0.1:7002> GET "is"
"you get the key 'is'"
```

而如果我们向节点7002发送关于键"love"的命令，那么客户端会先

被转向至节点7003，然后再次执行命令：

```
127.0.0.1:7002> GET "love"
-> Redirected to slot [16198] located at 127.0.0.1:7003
"you get the key 'love'"
127.0.0.1:7003>
```

被隐藏的ASK错误

和接到MOVED错误时的情况类似，集群模式的redis-cli在接到ASK错误时也不会打印错误，而是自动根据错误提供的IP地址和端口进行转向动作。如果想看到节点发送的ASK错误的话，可以使用单机模式的redis-cli客户端：

```
$ redis-cli -p 7002
127.0.0.1:7002> GET "love"
(error) ASK 16198 127.0.0.1:7003
```


注意

在写这篇文章的时候，集群模式的redis-cli并未支持ASK自动转向，上面展示的ASK自动转向行为实际上是根据MOVED自动转向行为虚构出来的。因此，当集群模式的redis-cli真正支持ASK自动转向时，它的行为和上面展示的行为可能会有所不同。

本节将对ASK错误的实现原理进行说明，并对比ASK错误和MOVED错误的区别。

17.5.1 CLUSTER SETSLOT IMPORTING命令的实现

clusterState结构的importing_slots_from数组记录了当前节点正在从其他节点导入的槽：

```
typedef struct clusterState {
  // ...
  clusterNode *importing_slots_from[16384];
  // ...
} clusterState;
```

如果importing_slots_from[i]的值不为NULL，而是指向一个

clusterNode结构，那么表示当前节点正在从clusterNode所代表的节点导入槽i。

在对集群进行重新分片的时候，向目标节点发送命令：

```
CLUSTER SETSLOT <i> IMPORTING <source_id>
```

可以将目标节点clusterState.importing_slots_from[i]的值设置为source_id所代表节点的clusterNode结构。

举个例子，如果客户端向节点7003发送以下命令：

```
# 9dfb...
是节点7002
的ID
127.0.0.1:7003> CLUSTER SETSLOT 16198 IMPORTING 9dfb4c4e016e627d9769e4c9bb0d4fa208e65
OK
```

那么节点7003的clusterState.importing_slots_from数组将变成图17-27所示的样子。

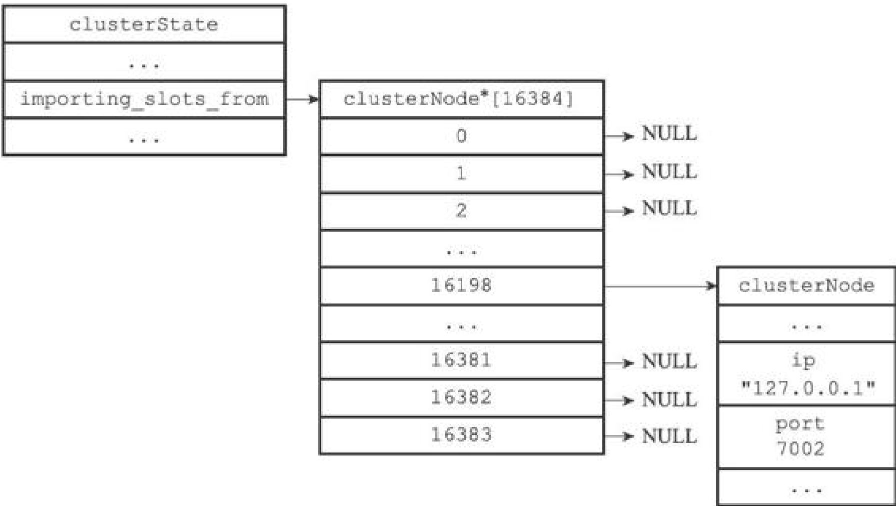


图17-27　节点7003的importing_slots_from数组

17.5.2　CLUSTER SETSLOT MIGRATING命令的实现

clusterState结构的migrating_slots_to数组记录了当前节点正在迁移至其他节点的槽：

```
typedef struct clusterState {
   // ...
   clusterNode *migrating_slots_to[16384];
   // ...
} clusterState;
```

如果migrating_slots_to[i]的值不为NULL，而是指向一个clusterNode结构，那么表示当前节点正在将槽i迁移至clusterNode所代表的节点。

在对集群进行重新分片的时候，向源节点发送命令：

```
CLUSTER SETSLOT <i> MIGRATING <target_id>
```

可以将源节点clusterState.migrating_slots_to[i]的值设置为target_id所代表节点的clusterNode结构。

举个例子，如果客户端向节点7002发送以下命令：

```
# 0457...
是节点7003
的ID
127.0.0.1:7002> CLUSTER SETSLOT 16198 MIGRATING 04579925484ce537d3410d7ce97bd2e260c45
OK
```

那么节点7002的clusterState.migrating_slots_to数组将变成图17-28所示的样子。

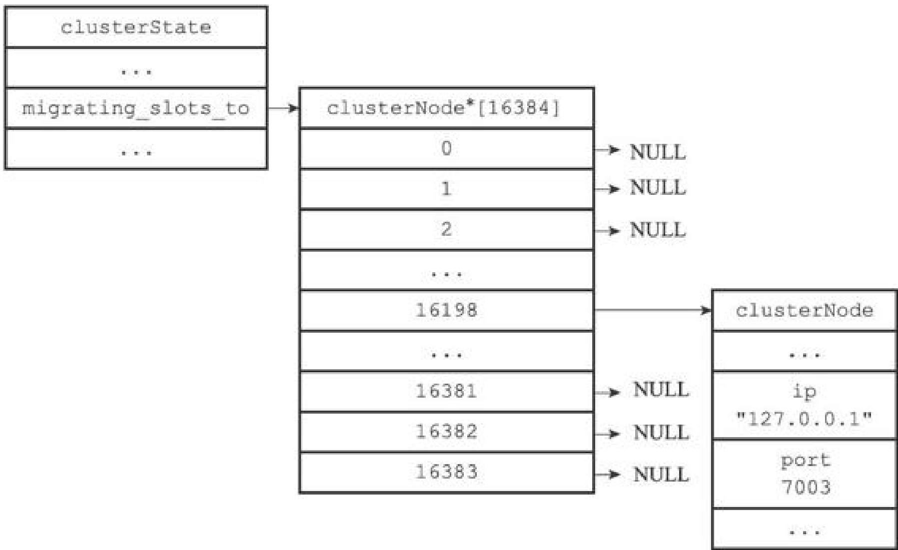


图17-28　节点7002的migrating_slots_to数组

17.5.3　ASK错误

如果节点收到一个关于键key的命令请求，并且键key所属的槽i正好就指派给了这个节点，那么节点会尝试在自己的数据库里查找键key，如果找到了的话，节点就直接执行客户端发送的命令。

与此相反，如果节点没有在自己的数据库里找到键key，那么节点

会检查自己的clusterState.migrating_slots_to[i]，看键key所属的槽i是否正在进行迁移，如果槽i的确在进行迁移的话，那么节点会向客户端发送一个ASK错误，引导客户端到正在导入槽i的节点去查找键key。

举个例子，假设在节点7002向节点7003迁移槽16198期间，有一个客户端向节点7002发送命令：

```
GET
"love
"
```

因为键"love"正好属于槽16198，所以节点7002会首先在自己的数据库中查找键"love"，但并没有找到，通过检查自己的clusterState.migrating_slots_to[16198]，节点7002发现自己正在将槽16198迁移至节点7003，于是它向客户端返回错误：

```
ASK 16198 127.0.0.1:7003
```

这个错误表示客户端可以尝试到IP为127.0.0.1，端口号为7003的节点去执行和槽16198有关的操作，如图17-29所示。

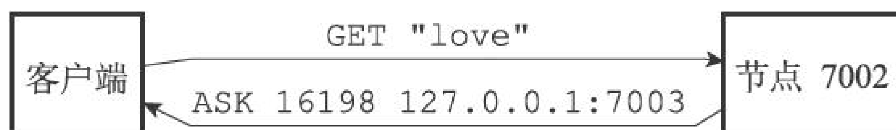


图17-29　客户端接收到节点7002返回的ASK错误

接到ASK错误的客户端会根据错误提供的IP地址和端口号，转向至正在导入槽的目标节点，然后首先向目标节点发送一个ASKING命令，之后再重新发送原本想要执行的命令。

以前面的例子来说，当客户端接收到节点7002返回的以下错误时：

```
ASK 16198 127.0.0.1:7003
```

客户端会转向至节点7003，首先发送命令：

```
ASKING
```

然后再次发送命令：

```
GET "love"
```

并获得回复：

```
"you get the key 'love'"
```

整个过程如图17-30所示。

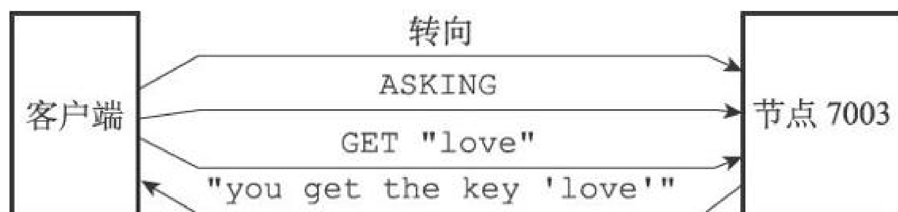


图17-30　客户端转向至节点7003

17.5.4　ASKING命令

ASKING命令唯一要做的就是打开发送该命令的客户端的REDIS_ASKING标识，以下是该命令的伪代码实现：

```
def ASKING():
    #
打开标识
    client.flags |= REDIS_ASKING
    #
向客户端返回OK
回复
    reply("OK")
```

在一般情况下，如果客户端向节点发送一个关于槽i的命令，而槽i又没有指派给这个节点的话，那么节点将向客户端返回一个MOVED错误；但是，如果节点的clusterState.importing_slots_from[i]显示节点正在导入槽i，并且发送命令的客户端带有REDIS_ASKING标识，那么节点将破例执行这个关于槽i的命令一次，图17-31展示了这个判断过程。

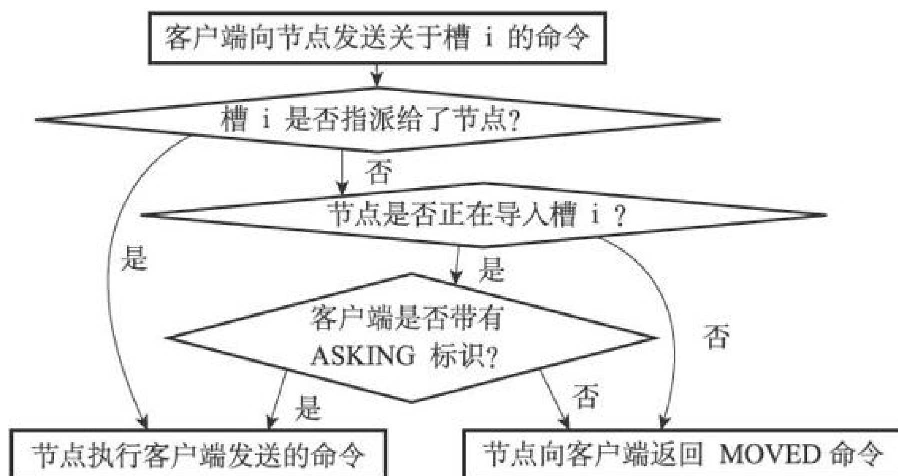


图17-31　节点判断是否执行客户端命令的过程

当客户端接收到ASK错误并转向至正在导入槽的节点时，客户端会先向节点发送一个ASKING命令，然后才重新发送想要执行的命令，这

是因为如果客户端不发送ASKING命令，而直接发送想要执行的命令的话，那么客户端发送的命令将被节点拒绝执行，并返回MOVED错误。

举个例子，我们可以使用普通模式的redis-cli客户端，向正在导入槽16198的节点7003发送以下命令：

```
$ ./redis-cli -p 7003
127.0.0.1:7003> GET "love"
(error) MOVED 16198 127.0.0.1:7002
```

虽然节点7003正在导入槽16198，但槽16198目前仍然是指派给了节点7002，所以节点7003会向客户端返回MOVED错误，指引客户端转向至节点7002。

但是，如果我们在发送GET命令之前，先向节点发送一个ASKING命令，那么这个GET命令就会被节点7003执行：

```
127.0.0.1:7003> ASKING
OK
127.0.0.1:7003> GET "love"
"you get the key 'love'"
```

另外要注意的是，客户端的REDIS_ASKING标识是一个一次性标识，当节点执行了一个带有REDIS_ASKING标识的客户端发送的命令之后，客户端的REDIS_ASKING标识就会被移除。

举个例子，如果我们在成功执行GET命令之后，再次向节点7003发送GET命令，那么第二次发送的GET命令将执行失败，因为这时客户端的REDIS_ASKING标识已经被移除：

```
127.0.0.1:7003> ASKING                  #
打开REDIS_ASKING
标识
OK
127.0.0.1:7003> GET "love"    #
移除REDIS_ASKING
标识
"you get the key 'love'"
127.0.0.1:7003> GET "love"    # REDIS_ASKING
标识未打开，执行失败
(error) MOVED 16198 127.0.0.1:7002
```

17.5.5　ASK错误和MOVED错误的区别

ASK错误和MOVED错误都会导致客户端转向，它们的区别在于：

·MOVED错误代表槽的负责权已经从一个节点转移到了另一个节点：在客户端收到关于槽i的MOVED错误之后，客户端每次遇到关于槽i的命令请求时，都可以直接将命令请求发送至MOVED错误所指向的节点，因为该节点就是目前负责槽i的节点。

·与此相反，ASK错误只是两个节点在迁移槽的过程中使用的一种临时措施：在客户端收到关于槽i的ASK错误之后，客户端只会在接下来的一次命令请求中将关于槽i的命令请求发送至ASK错误所指示的节点，但这种转向不会对客户端今后发送关于槽i的命令请求产生任何影响，客户端仍然会将关于槽i的命令请求发送至目前负责处理槽i的节点，除非ASK错误再次出现。

17.6 复制与故障转移

Redis集群中的节点分为主节点（master）和从节点（slave），其中主节点用于处理槽，而从节点则用于复制某个主节点，并在被复制的主节点下线时，代替下线主节点继续处理命令请求。

举个例子，对于包含7000、7001、7002、7003四个主节点的集群来说，我们可以将7004、7005两个节点添加到集群里面，并将这两个节点设定为节点7000的从节点，如图17-32所示（图中以双圆形表示主节点，单圆形表示从节点）。

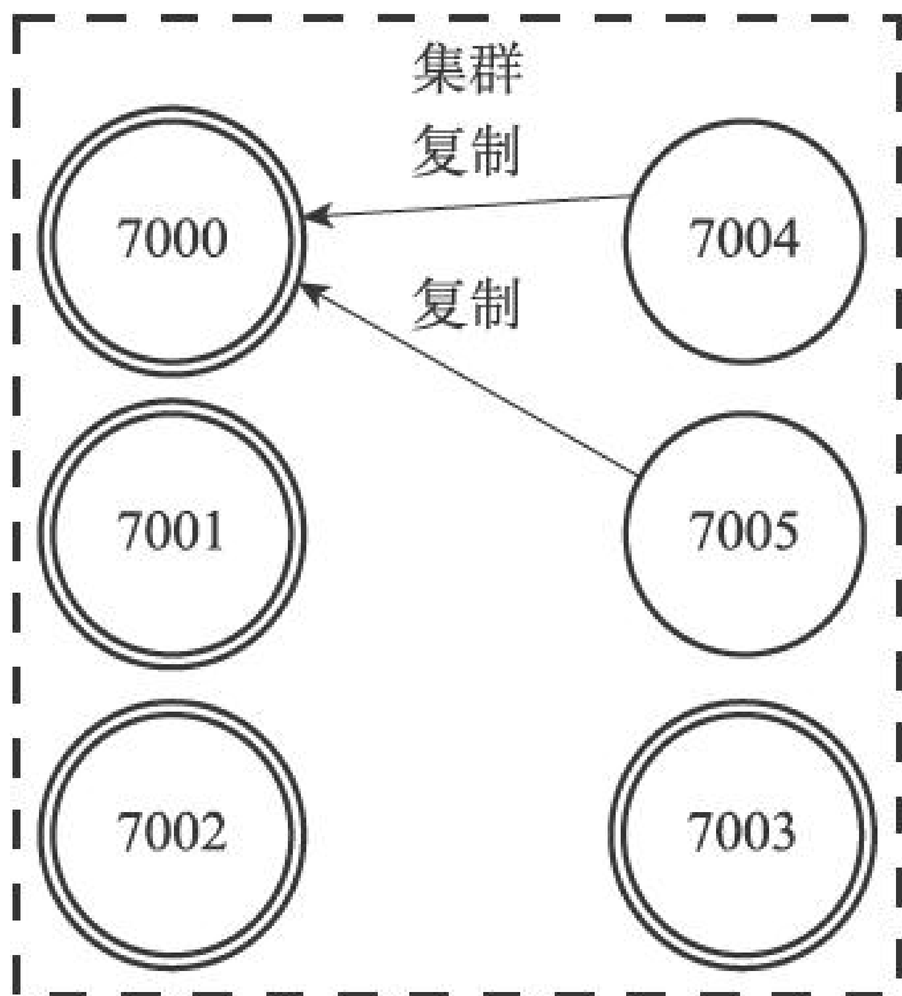


图17-32　设置节点7004和节点7005成为节点7000的从节点

表17-1记录了集群各个节点的当前状态，以及它们正在做的工作。

表17-1　集群各个节点的当前状态

节点	角色	状态	工作
7000	主节点	在线	负责处理槽 0 至槽 5000
7001	主节点	在线	负责处理槽 5001 至槽 10000
7002	主节点	在线	负责处理槽 10001 至槽 15000
7003	主节点	在线	负责处理槽 15001 至槽 16383
7004	从节点	在线	复制节点 7000
7005	从节点	在线	复制节点 7000

如果这时，节点7000进入下线状态，那么集群中仍在正常运作的几个主节点将在节点7000的两个从节点——节点7004和节点7005中选出一个节点作为新的主节点，这个新的主节点将接管原来节点7000负责处理的槽，并继续处理客户端发送的命令请求。

例如，如果节点7004被选中为新的主节点，那么节点7004将接管原来由节点7000负责处理的槽0至槽5000，节点7005也会从原来的复制节点7000，改为复制节点7004，如图17-33所示（图中用虚线包围的节点为已下线节点）。

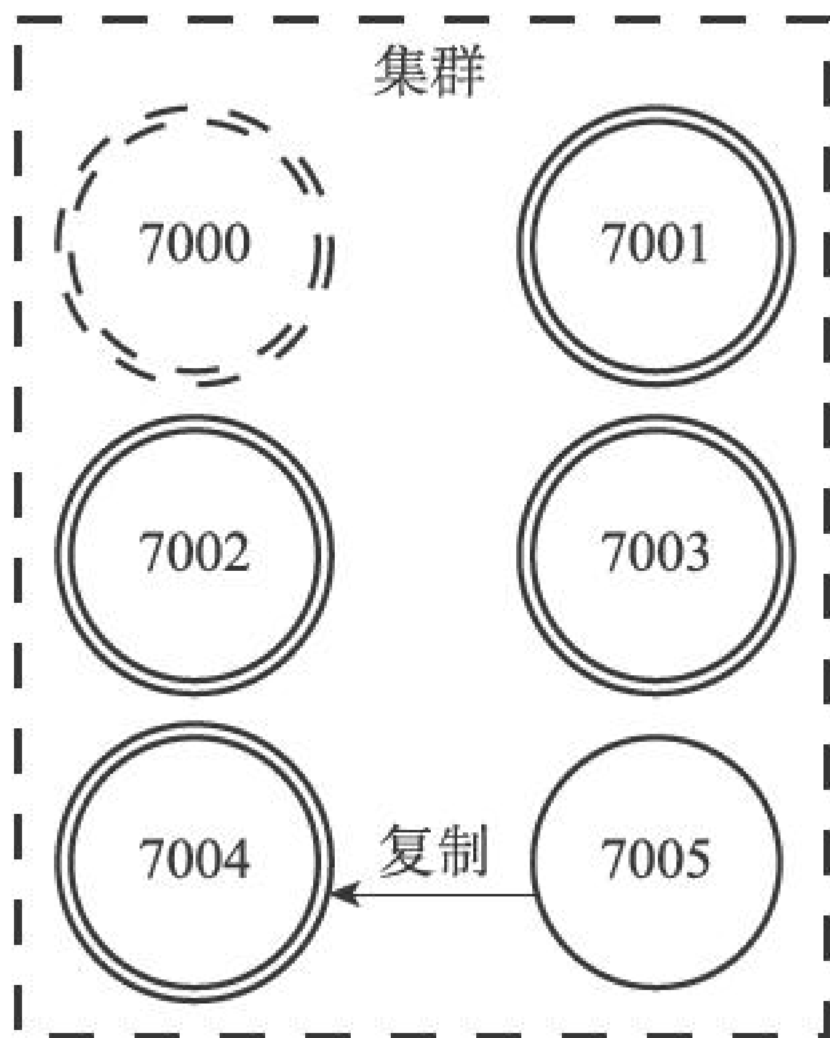


图17-33　节点7004成为新的主节点

表17-2记录了在对节点7000进行故障转移之后，集群各个节点的当前状态，以及它们正在做的工作。

表17-2　集群各个节点的当前状态

节点	角色	状态	工作
7000	主节点	下线	负责处理槽 0 至槽 5000（因为故障转移已经完成，所以该工作已经无效。）
7001	主节点	在线	负责处理槽 5001 至槽 10000
7002	主节点	在线	负责处理槽 10001 至槽 15000
7003	主节点	在线	负责处理槽 15001 至槽 16383
7004	主节点	在线	负责处理槽 0 至槽 5000
7005	从节点	在线	复制节点 7004

如果在故障转移完成之后，下线的节点7000重新上线，那么它将成为节点7004的从节点，如图17-34所示。

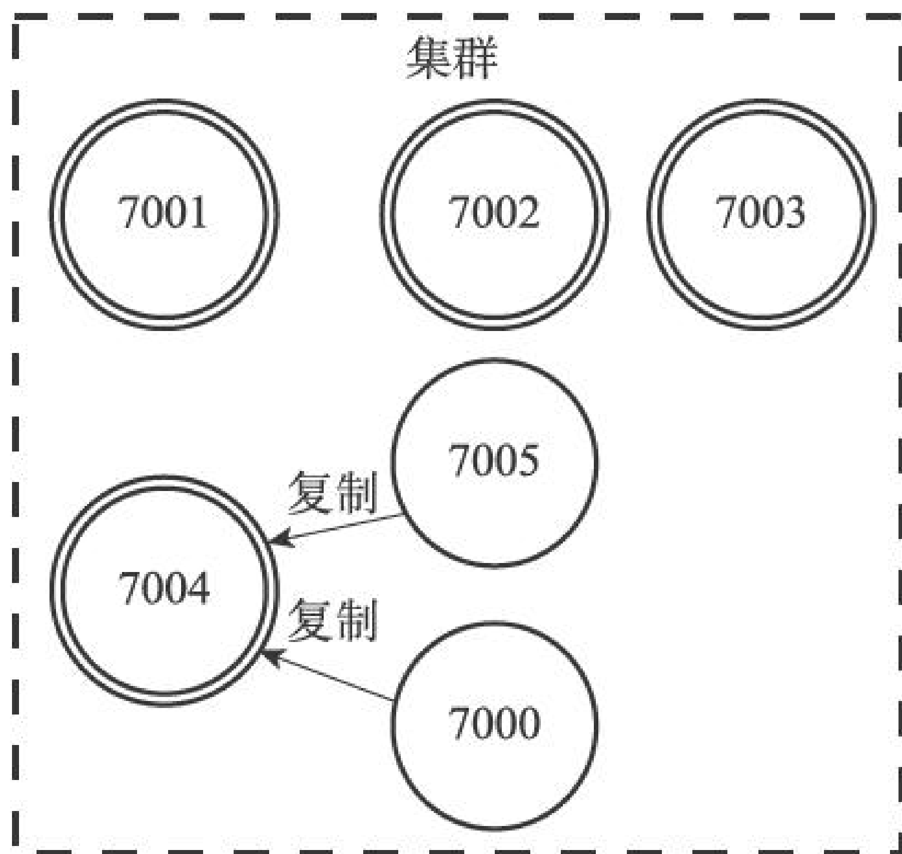


图17-34　重新上线的节点7000成为节点7004的从节点

表17-3展示了节点7000复制节点7004之后，集群中各个节点的状态。

表17-3　集群各个节点的当前状态

节点	角色	状态	工作
7000	从节点	在线	复制节点 7004
7001	主节点	在线	负责处理槽 5001 至槽 10000
7002	主节点	在线	负责处理槽 10001 至槽 15000
7003	主节点	在线	负责处理槽 15001 至槽 16383
7004	主节点	在线	负责处理槽 0 至槽 5000
7005	从节点	在线	复制节点 7004

本节接下来的内容将介绍节点的复制方法，检测节点是否下线的方法，以及对下线主节点进行故障转移的方法。

17.6.1　设置从节点

向一个节点发送命令：

```
CLUSTER REPLICATE <node_id>
```

可以让接收命令的节点成为node_id所指定节点的从节点，并开始对主节点进行复制：

·接收到该命令的节点首先会在自己的clusterState.nodes字典中找到node_id所对应节点的clusterNode结构，并将自己的clusterState.myself.slaveof指针指向这个结构，以此来记录这个节点正在复制的主节点：

```
struct clusterNode {
  // ...
  //
如果这是一个从节点，那么指向主节点
  struct clusterNode *slaveof;
  // ...
};
```

·然后节点会修改自己在clusterState.myself.flags中的属性，关闭原本的REDIS_NODE_MASTER标识，打开REDIS_NODE_SLAVE标识，表示这个节点已经由原来的主节点变成了从节点。

·最后，节点会调用复制代码，并根据clusterState.myself.slaveof指向的clusterNode结构所保存的IP地址和端口号，对主节点进行复制。因为节点的复制功能和单机Redis服务器的复制功能使用了相同的代码，所以让从节点复制主节点相当于向从节点发送命令SLAVEOF。

图17-35展示了节点7004在复制节点7000时的clusterState结构：

·clusterState.myself.flags属性的值为REDIS_NODE_SLAVE，表示节点7004是一个从节点。

·clusterState.myself.slaveof指针指向代表节点7000的结构，表示节点7004正在复制的主节点为节点7000。

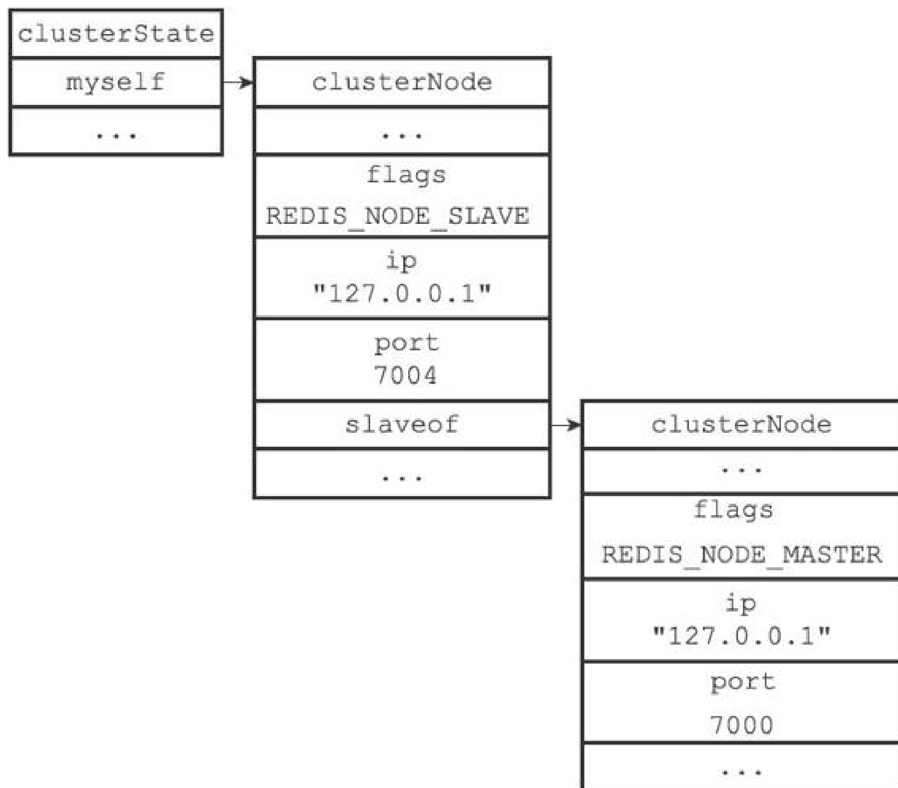


图17-35　节点7004的clusterState结构

一个节点成为从节点，并开始复制某个主节点这一信息会通过消息发送给集群中的其他节点，最终集群中的所有节点都会知道某个从节点正在复制某个主节点。

集群中的所有节点都会在代表主节点的clusterNode结构的slaves属性和numslaves属性中记录正在复制这个主节点的从节点名单：

```
struct clusterNode {
    // ...
    //
正在复制这个主节点的从节点数量
    int numslaves;
    //
一个数组
    //
每个数组项指向一个正在复制这个主节点的从节点的clusterNode
结构
    struct clusterNode **slaves;
    // ...
};
```

举个例子，图17-36记录了节点7004和节点7005成为节点7000的从节点之后，集群中的各个节点为节点7000创建的clusterNode结构的样子：

·代表节点7000的clusterNode结构的numslaves属性的值为2，这说明有两个从节点正在复制节点7000。

·代表节点7000的clusterNode结构的slaves数组的两个项分别指向代表节点7004和代表节点7005的clusterNode结构，这说明节点7000的两个从节点分别是节点7004和节点7005。

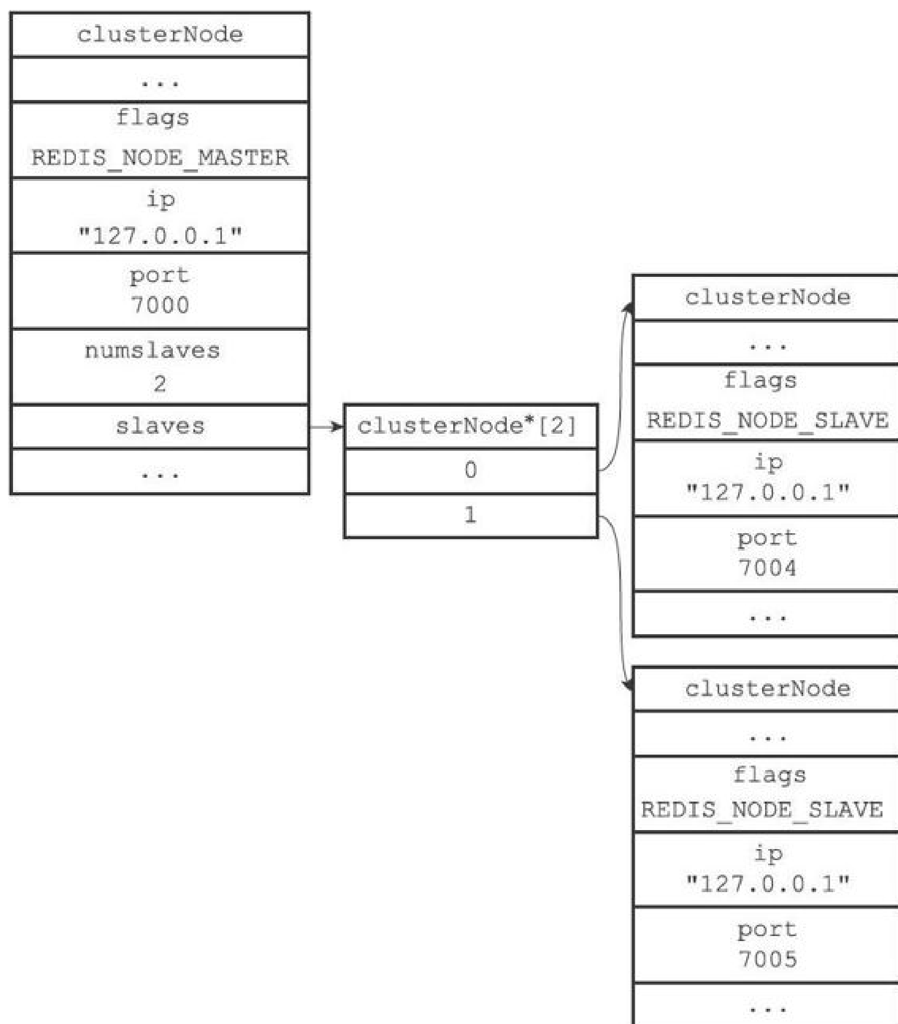


图17-36　集群中的各个节点为节点7000创建的clusterNode结构

17.6.2　故障检测

集群中的每个节点都会定期地向集群中的其他节点发送PING消息，以此来检测对方是否在线，如果接收PING消息的节点没有在规定的时间内，向发送PING消息的节点返回PONG消息，那么发送PING消息的节点就会将接收PING消息的节点标记为疑似下线（probable fail，

PFAIL）。

举个例子，如果节点7001向节点7000发送了一条PING消息，但是节点7000没有在规定的时间内，向节点7001返回一条PONG消息，那么节点7001就会在自己的clusterState.nodes字典中找到节点7000所对应的clusterNode结构，并在结构的flags属性中打开REDIS_NODE_PFAIL标识，以此表示节点7000进入了疑似下线状态，如图17-37所示。

clusterNode
...
flags REDIS_NODE_MASTER & REDIS_NODE_PFAIL
ip "127.0.0.1"
port 7000
...

图17-37　代表节点7000的clusterNode结构

集群中的各个节点会通过互相发送消息的方式来交换集群中各个节点的状态信息，例如某个节点是处于在线状态、疑似下线状态（PFAIL），还是已下线状态（FAIL）。

当一个主节点A通过消息得知主节点B认为主节点C进入了疑似下线状态时，主节点A会在自己的clusterState.nodes字典中找到主节点C所对应的clusterNode结构，并将主节点B的下线报告（failure　report）添加到clusterNode结构的fail_reports链表里面：

```
struct clusterNode {
  // ...
  //
一个链表，记录了所有其他节点对该节点的下线报告
  list *fail_reports;
  // ...
};
```

每个下线报告由一个clusterNodeFailReport结构表示：

```
struct clusterNodeFailReport {
  //
报告目标节点已经下线的节点
  struct clusterNode *node;
  //
最后一次从node
节点收到下线报告的时间
```

```
    //
程序使用这个时间戳来检查下线报告是否过期
    //
（与当前时间相差太久的下线报告会被删除）
    mstime_t time;
} typedef clusterNodeFailReport;
```

举个例子，如果主节点7001在收到主节点7002、主节点7003发送的消息后得知，主节点7002和主节点7003都认为主节点7000进入了疑似下线状态，那么主节点7001将为主节点7000创建图17-38所示的下线报告。

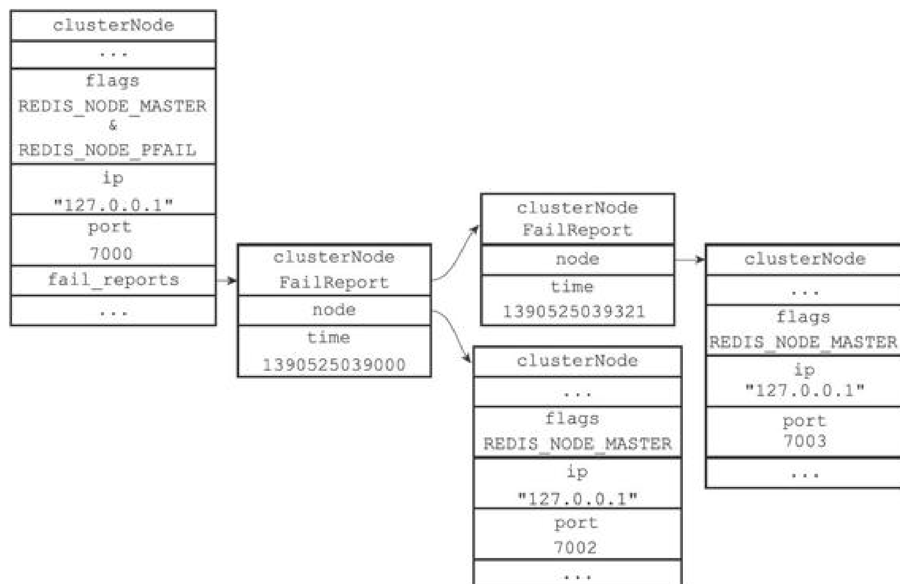


图17-38　节点7000的下线报告

如果在一个集群里面，半数以上负责处理槽的主节点都将某个主节点x报告为疑似下线，那么这个主节点x将被标记为已下线（FAIL），将主节点x标记为已下线的节点会向集群广播一条关于主节点x的FAIL消息，所有收到这条FAIL消息的节点都会立即将主节点x标记为已下线。

举个例子，对于图17-38所示的下线报告来说，主节点7002和主节点7003都认为主节点7000进入了下线状态，并且主节点7001也认为主节点7000进入了疑似下线状态（代表主节点7000的结构打开了REDIS_NODE_PFAIL标识），综合起来，在集群四个负责处理槽的主节点里面，有三个都将主节点7000标记为下线，数量已经超过了半数，所以主节点7001会将主节点7000标记为已下线，并向集群广播一条关于主节点7000的FAIL消息，如图17-39所示。

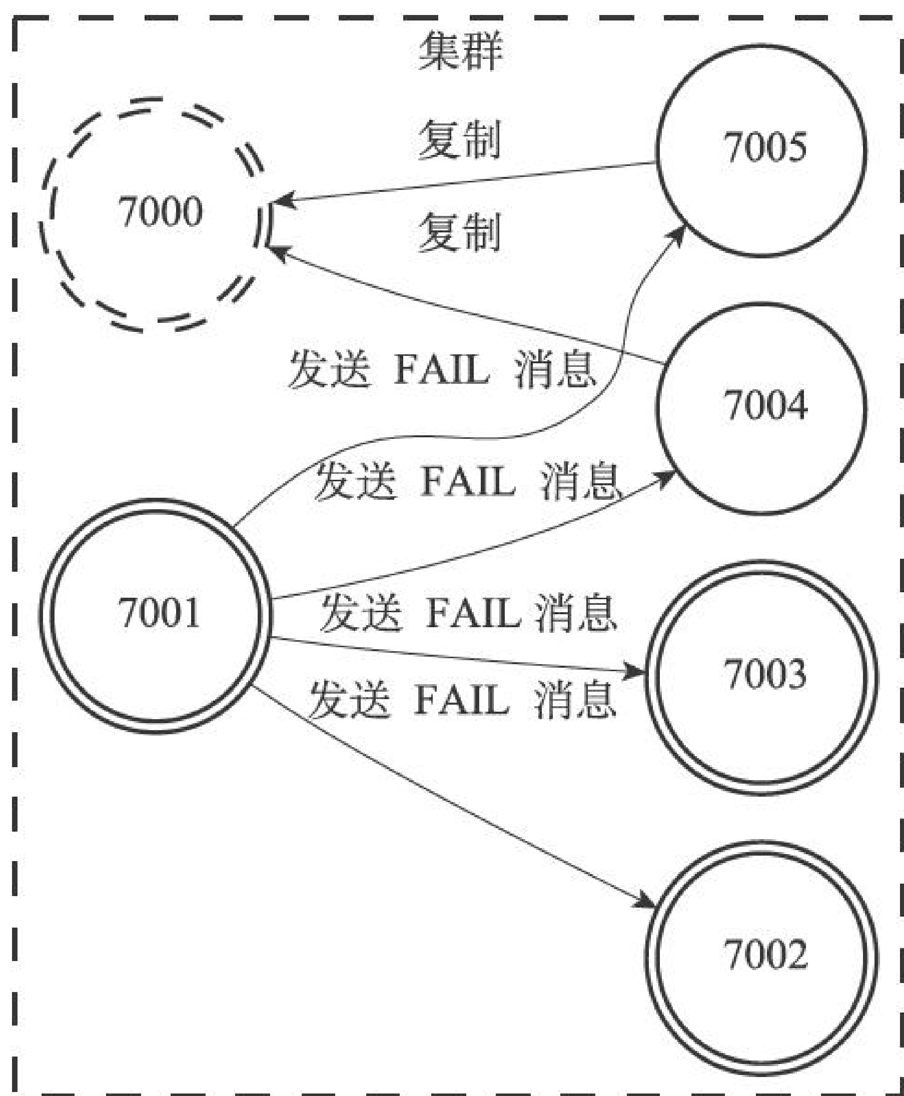


图17-39　节点7001向集群广播FAIL消息

17.6.3　故障转移

当一个从节点发现自己正在复制的主节点进入了已下线状态时，从节点将开始对下线主节点进行故障转移，以下是故障转移的执行步骤：

1）复制下线主节点的所有从节点里面，会有一个从节点被选中。

2）被选中的从节点会执行SLAVEOF　no　one命令，成为新的主节点。

3）新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己。

4）新的主节点向集群广播一条PONG消息，这条PONG消息可以让集群中的其他节点立即知道这个节点已经由从节点变成了主节点，并且这个主节点已经接管了原本由已下线节点负责处理的槽。

5）新的主节点开始接收和自己负责处理的槽有关的命令请求，故障转移完成。

17.6.4　选举新的主节点

新的主节点是通过选举产生的。

以下是集群选举新的主节点的方法：

1）集群的配置纪元是一个自增计数器，它的初始值为0。

2）当集群里的某个节点开始一次故障转移操作时，集群配置纪元的值会被增一。

3）对于每个配置纪元，集群里每个负责处理槽的主节点都有一次投票的机会，而第一个向主节点要求投票的从节点将获得主节点的投票。

4）当从节点发现自己正在复制的主节点进入已下线状态时，从节点会向集群广播一条CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST消息，要求所有收到这条消息、并且具有投票权的主节点向这个从节点投票。

5）如果一个主节点具有投票权（它正在负责处理槽），并且这个主节点尚未投票给其他从节点，那么主节点将向要求投票的从节点返回一条CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK消息，表示这个主节点支持从节点成为新的主节点。

6）每个参与选举的从节点都会接收CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK消息，并根据自己收到了多少条这种消息来统计自己获得了多少主节点的支持。

7）如果集群里有N个具有投票权的主节点，那么当一个从节点收集到大于等于N/2+1张支持票时，这个从节点就会当选为新的主节点。

8）因为在每一个配置纪元里面，每个具有投票权的主节点只能投一次票，所以如果有N个主节点进行投票，那么具有大于等于N/2+1张支持票的从节点只会有一个，这确保了新的主节点只会有一个。

9）如果在一个配置纪元里面没有从节点能收集到足够多的支持票，那么集群进入一个新的配置纪元，并再次进行选举，直到选出新的主节点为止。

这个选举新主节点的方法和第16章介绍的选举领头Sentinel的方法非常相似，因为两者都是基于Raft算法的领头选举（leader election）方法来实现的。

17.7　消息

集群中的各个节点通过发送和接收消息（message）来进行通信，我们称发送消息的节点为发送者（sender），接收消息的节点为接收者（receiver），如图17-40所示。

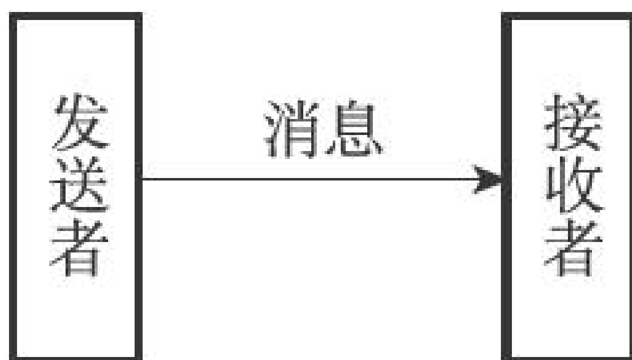


图17-40　发送者和接收者

节点发送的消息主要有以下五种：

·MEET消息：当发送者接到客户端发送的CLUSTER　MEET命令时，发送者会向接收者发送MEET消息，请求接收者加入到发送者当前所处的集群里面。

·PING消息：集群里的每个节点默认每隔一秒钟就会从已知节点列表中随机选出五个节点，然后对这五个节点中最长时间没有发送过PING消息的节点发送PING消息，以此来检测被选中的节点是否在线。除此之外，如果节点A最后一次收到节点B发送的PONG消息的时间，距离当前时间已经超过了节点A的cluster-node-timeout选项设置时长的一半，那么节点A也会向节点B发送PING消息，这可以防止节点A因为长时间没有随机选中节点B作为PING消息的发送对象而导致对节点B的信息更新滞后。

·PONG消息：当接收者收到发送者发来的MEET消息或者PING消息时，为了向发送者确认这条MEET消息或者PING消息已到达，接收者会向发送者返回一条PONG消息。另外，一个节点也可以通过向集群广播自己的PONG消息来让集群中的其他节点立即刷新关于这个节点的认识，例如当一次故障转移操作成功执行之后，新的主节点会向集群广播一条PONG消息，以此来让集群中的其他节点立即知道这个节点已经变成了主节点，并且接管了已下线节点负责的槽。

·FAIL消息：当一个主节点A判断另一个主节点B已经进入FAIL状态时，节点A会向集群广播一条关于节点B的FAIL消息，所有收到这条消息的节点都会立即将节点B标记为已下线。

·PUBLISH消息：当节点接收到一个PUBLISH命令时，节点会执行这个命令，并向集群广播一条PUBLISH消息，所有接收到这条PUBLISH消息的节点都会执行相同的PUBLISH命令。

一条消息由消息头（header）和消息正文（data）组成，接下来的内容将首先介绍消息头，然后再分别介绍上面提到的五种不同类型的消息正文。

17.7.1 消息头

节点发送的所有消息都由一个消息头包裹，消息头除了包含消息正文之外，还记录了消息发送者自身的一些信息，因为这些信息也会被消息接收者用到，所以严格来讲，我们可以认为消息头本身也是消息的一部分。

每个消息头都由一个cluster.h/clusterMsg结构表示：

```
typedef struct {
  //
消息的长度（包括这个消息头的长度和消息正文的长度）
  uint32_t totlen;
  //
消息的类型
  uint16_t type;
  //
消息正文包含的节点信息数量
  //
只在发送MEET
、PING
、PONG
这三种Gossip
协议消息时使用
  uint16_t count;
  //
发送者所处的配置纪元
  uint64_t currentEpoch;
  //
如果发送者是一个主节点，那么这里记录的是发送者的配置纪元
  //
如果发送者是一个从节点，那么这里记录的是发送者正在复制的主节点的配置纪元
  uint64_t configEpoch;
  //
发送者的名字（ID
）
  char sender[REDIS_CLUSTER_NAMELEN];
  //
发送者目前的槽指派信息
  unsigned char myslots[REDIS_CLUSTER_SLOTS/8];
  //
如果发送者是一个从节点，那么这里记录的是发送者正在复制的主节点的名字
  //
如果发送者是一个主节点，那么这里记录的是REDIS_NODE_NULL_NAME
  //
（一个40
字节长，值全为0
的字节数组）
  char slaveof[REDIS_CLUSTER_NAMELEN];
  //
发送者的端口号
  uint16_t port;
  //
```

```
发送者的标识值
  uint16_t flags;
  //
发送者所处集群的状态
  unsigned char state;
  //
消息的正文（或者说，内容）
  union clusterMsgData data;
} clusterMsg;
```

clusterMsg.data属性指向联合cluster.h/clusterMsgData，这个联合就是消息的正文：

```
union clusterMsgData {
  // MEET
、PING
、PONG
消息的正文
  struct {
    //
每条MEET
、PING
、PONG
消息都包含两个
    // clusterMsgDataGossip
结构
    clusterMsgDataGossip gossip[1];
  } ping;
  // FAIL
消息的正文
  struct {
    clusterMsgDataFail about;
  } fail;
  // PUBLISH
消息的正文
  struct {
    clusterMsgDataPublish msg;
  } publish;
  //
其他消息的正文...
};
```

clusterMsg结构的currentEpoch、sender、myslots等属性记录了发送者自身的节点信息，接收者会根据这些信息，在自己的clusterState.nodes字典里找到发送者对应的clusterNode结构，并对结构进行更新。

举个例子，通过对比接收者为发送者记录的槽指派信息，以及发送者在消息头的myslots属性记录的槽指派信息，接收者可以知道发送者的槽指派信息是否发生了变化。

又或者说，通过对比接收者为发送者记录的标识值，以及发送者在消息头的flags属性记录的标识值，接收者可以知道发送者的状态和角色是否发生了变化，例如节点状态由原来的在线变成了下线，或者由主节点变成了从节点等等。

17.7.2 MEET、PING、PONG消息的实现

Redis集群中的各个节点通过Gossip协议来交换各自关于不同节点的状态信息，其中Gossip协议由MEET、PING、PONG三种消息实现，这三种消息的正文都由两个cluster.h/clusterMsgDataGossip结构组成：

```
union clusterMsgData {
  // ...
  // MEET
、PING
和PONG
消息的正文
  struct {
    //
每条MEET
、PING
、PONG
消息都包含两个
    // clusterMsgDataGossip
结构
    clusterMsgDataGossip gossip[1];
  } ping;
  //
其他消息的正文...
};
```

因为MEET、PING、PONG三种消息都使用相同的消息正文，所以节点通过消息头的type属性来判断一条消息是MEET消息、PING消息还是PONG消息。

每次发送MEET、PING、PONG消息时，发送者都从自己的已知节点列表中随机选出两个节点（可以是主节点或者从节点），并将这两个被选中节点的信息分别保存到两个clusterMsgDataGossip结构里面。

clusterMsgDataGossip结构记录了被选中节点的名字，发送者与被选中节点最后一次发送和接收PING消息和PONG消息的时间戳，被选中节点的IP地址和端口号，以及被选中节点的标识值：

```
typedef struct {
  //
节点的名字
  char nodename[REDIS_CLUSTER_NAMELEN];
  //
最后一次向该节点发送PING
消息的时间戳
  uint32_t ping_sent;
  //
最后一次从该节点接收到PONG
消息的时间戳
  uint32_t pong_received;
  //
节点的IP
地址
  char ip[16];
  //
节点的端口号
  uint16_t port;
  //
节点的标识值
  uint16_t flags;
} clusterMsgDataGossip;
```

当接收者收到MEET、PING、PONG消息时，接收者会访问消息正文中的两个clusterMsgDataGossip结构，并根据自己是否认识clusterMsgDataGossip结构中记录的被选中节点来选择进行哪种操作：

·如果被选中节点不存在于接收者的已知节点列表，那么说明接收者是第一次接触到被选中节点，接收者将根据结构中记录的IP地址和端口号等信息，与被选中节点进行握手。

·如果被选中节点已经存在于接收者的已知节点列表，那么说明接收者之前已经与被选中节点进行过接触，接收者将根据clusterMsgDataGossip结构记录的信息，对被选中节点所对应的clusterNode结构进行更新。

举个发送PING消息和返回PONG消息的例子，假设在一个包含A、B、C、D、E、F六个节点的集群里：

·节点A向节点D发送PING消息，并且消息里面包含了节点B和节点C的信息，当节点D收到这条PING消息时，它将更新自己对节点B和节点C的认识。

·之后，节点D将向节点A返回一条PONG消息，并且消息里面包含了节点E和节点F的消息，当节点A收到这条PONG消息时，它将更新自己对节点E和节点F的认识。

整个通信过程如图17-41所示。

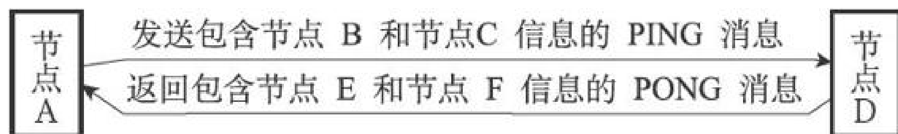


图17-41　一个PING-PONG消息通信示例

17.7.3　FAIL消息的实现

当集群里的主节点A将主节点B标记为已下线（FAIL）时，主节点A将向集群广播一条关于主节点B的FAIL消息，所有接收到这条FAIL消息的节点都会将主节点B标记为已下线。

在集群的节点数量比较大的情况下，单纯使用Gossip协议来传播节点的已下线信息会给节点的信息更新带来一定延迟，因为Gossip协议消息通常需要一段时间才能传播至整个集群，而发送FAIL消息可以让集群里的所有节点立即知道某个主节点已下线，从而尽快判断是否需要将集群标记为下线，又或者对下线主节点进行故障转移。

FAIL消息的正文由cluster.h/clusterMsgDataFail结构表示，这个结构只包含一个nodename属性，该属性记录了已下线节点的名字：

```
typedef struct {
    char nodename[REDIS_CLUSTER_NAMELEN];
} clusterMsgDataFail;
```

因为集群里的所有节点都有一个独一无二的名字，所以FAIL消息里面只需要保存下线节点的名字，接收到消息的节点就可以根据这个名字来判断是哪个节点下线了。

举个例子，对于包含7000、7001、7002、7003四个主节点的集群来说：

·如果主节点7001发现主节点7000已下线，那么主节点7001将向主节点7002和主节点7003发送FAIL消息，其中FAIL消息中包含的节点名字为主节点7000的名字，以此来表示主节点7000已下线。

·当主节点7002和主节点7003都接收到主节点7001发送的FAIL消息时，它们也会将主节点7000标记为已下线。

·因为这时集群已经有超过一半的主节点认为主节点7000已下线，所以集群剩下的几个主节点可以判断是否需要将集群标记为下线，又或者开始对主节点7000进行故障转移。

图17-42至图17-44展示了节点发送和接收FAIL消息的整个过程。

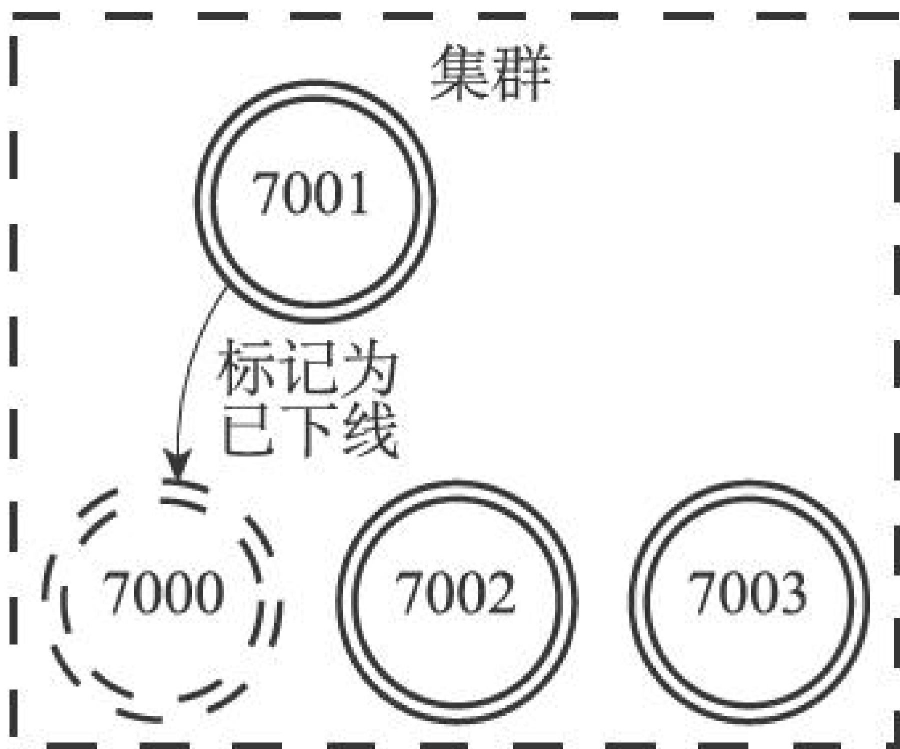


图17-42　节点7001将节点7000标记为已下线

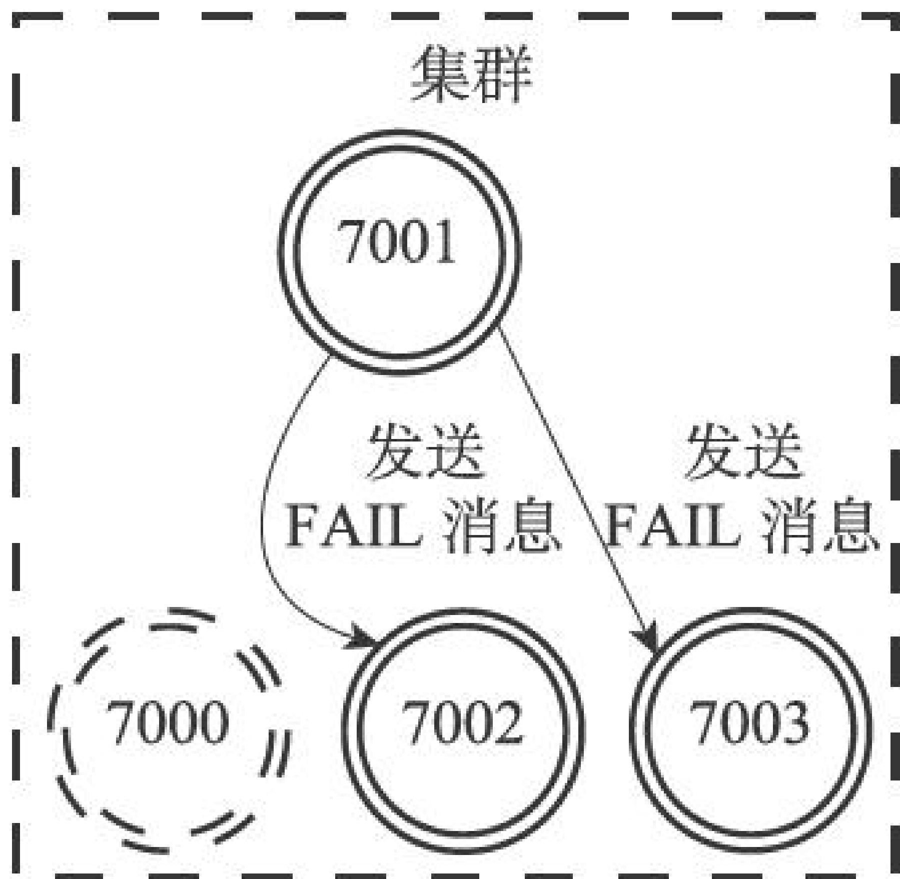


图17-43　节点7001向集群广播FAIL消息

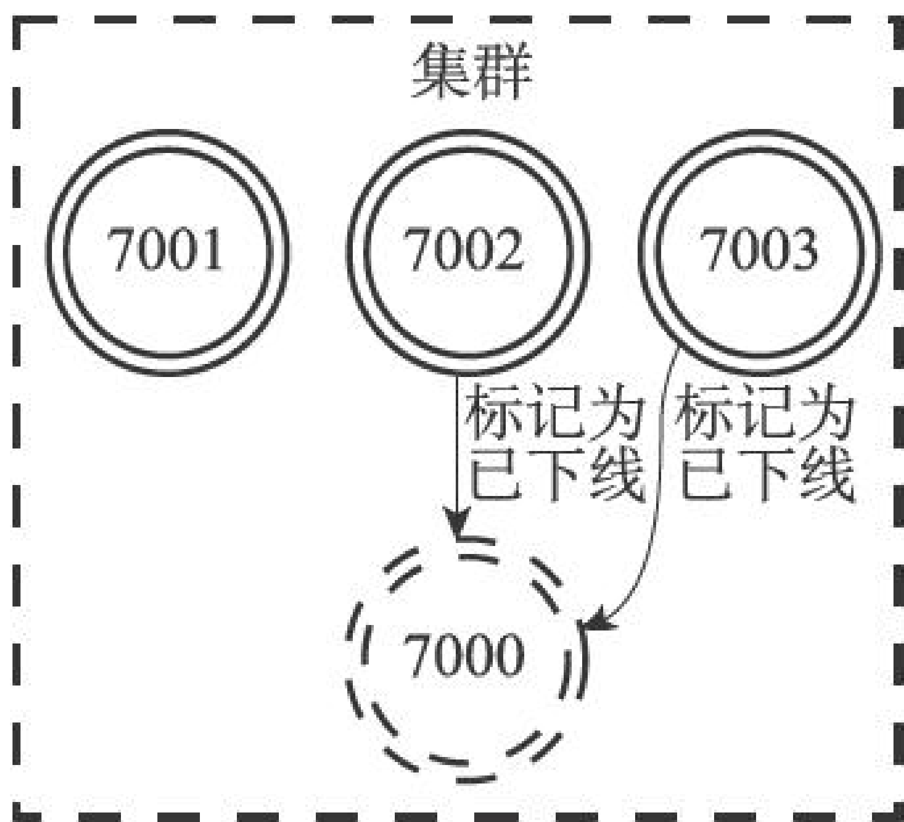


图17-44　节点7002和节点7003也将节点7000标记为已下线

17.7.4　PUBLISH消息的实现

当客户端向集群中的某个节点发送命令：

```
PUBLISH <channel> <message>
```

的时候，接收到PUBLISH命令的节点不仅会向channel频道发送消息message，它还会向集群广播一条PUBLISH消息，所有接收到这条PUBLISH消息的节点都会向channel频道发送message消息。

换句话说，向集群中的某个节点发送命令：

```
PUBLISH <channel> <message>
```

将导致集群中的所有节点都向channel频道发送message消息。

举个例子，对于包含7000、7001、7002、7003四个节点的集群来说，如果节点7000收到了客户端发送的PUBLISH命令，那么节点7000将向7001、7002、7003三个节点发送PUBLISH消息，如图17-45所示。

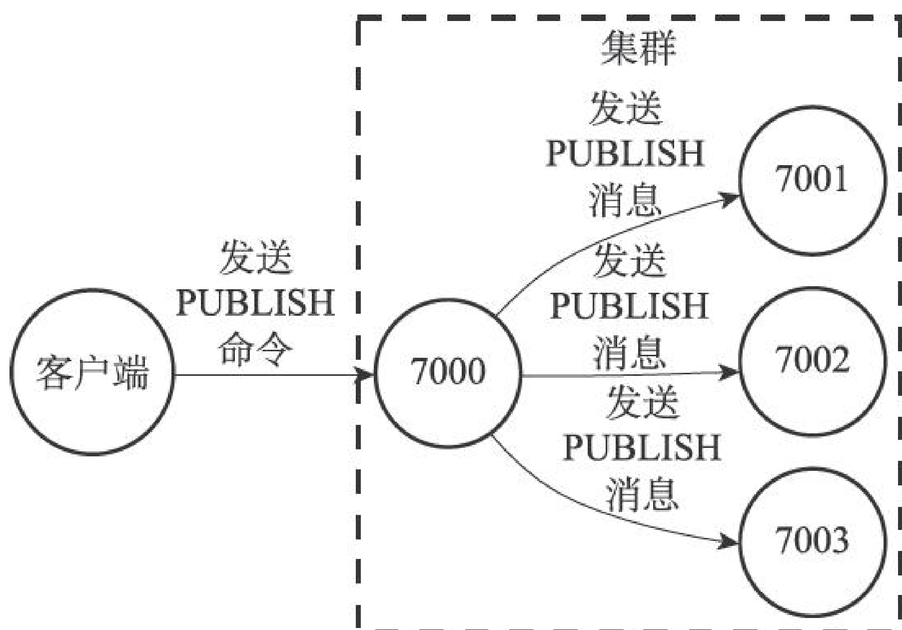


图17-45　接收到PUBLISH命令的节点7000向集群广播PUBLISH消息

PUBLISH消息的正文由cluster.h/clusterMsgDataPublish结构表示：

```
typedef struct {
  uint32_t channel_len;
  uint32_t message_len;
  //
定义为8
字节只是为了对齐其他消息结构
  //
实际的长度由保存的内容决定
  unsigned char bulk_data[8];
} clusterMsgDataPublish;
```

clusterMsgDataPublish结构的bulk_data属性是一个字节数组，这个字节数组保存了客户端通过PUBLISH命令发送给节点的channel参数和message参数，而结构的channel_len和message_len则分别保存了channel参数的长度和message参数的长度：

·其中bulk_data的0字节至channel_len-1字节保存的是channel参数。

·而bulk_data的channel_len字节至channel_len＋message_len-1字节保存的则是message参数。

举个例子，如果节点收到的PUBLISH命令为：

```
PUBLISH "news.it" "hello"
```

那么节点发送的PUBLISH消息的clusterMsgDataPublish结构将如图

17-46所示：其中bulk_data数组的前七个字节保存了channel参数的值"news.it"，而bulk_data数组的后五个字节则保存了message参数的值"hello"。

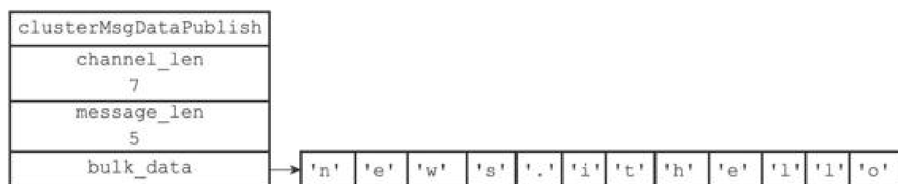


图17-46　clusterMsgDataPublish结构示例

为什么不直接向节点广播PUBLISH命令

实际上，要让集群的所有节点都执行相同的PUBLISH命令，最简单的方法就是向所有节点广播相同的PUBLISH命令，这也是Redis在复制PUBLISH命令时所使用的方法，不过因为这种做法并不符合Redis集群的“各个节点通过发送和接收消息来进行通信”这一规则，所以节点没有采取广播PUBLISH命令的做法。

17.8 重点回顾

·节点通过握手来将其他节点添加到自己所处的集群当中。

·集群中的16384个槽可以分别指派给集群中的各个节点，每个节点都会记录哪些槽指派给了自己，而哪些槽又被指派给了其他节点。

·节点在接到一个命令请求时，会先检查这个命令请求要处理的键所在的槽是否由自己负责，如果不是的话，节点将向客户端返回一个MOVED错误，MOVED错误携带的信息可以指引客户端转向至正在负责相关槽的节点。

·对Redis集群的重新分片工作是由redis-trib负责执行的，重新分片的关键是将属于某个槽的所有键值对从一个节点转移至另一个节点。

·如果节点A正在迁移槽i至节点B，那么当节点A没能在自己的数据库中找到命令指定的数据库键时，节点A会向客户端返回一个ASK错误，指引客户端到节点B继续查找指定的数据库键。

·MOVED错误表示槽的负责权已经从一个节点转移到了另一个节点，而ASK错误只是两个节点在迁移槽的过程中使用的一种临时措施。

·集群里的从节点用于复制主节点，并在主节点下线时，代替主节点继续处理命令请求。

·集群中的节点通过发送和接收消息来进行通信，常见的消息包括MEET、PING、PONG、PUBLISH、FAIL五种。

第四部分　独立功能的实现

第18章　发布与订阅

第19章　事务

第20章　Lua脚本

第21章　排序

第22章　二进制位数组

第23章　慢查询日志

第24章　监视器

第18章　发布与订阅

Redis的发布与订阅功能由PUBLISH、SUBSCRIBE、PSUBSCRIBE等命令组成。

通过执行SUBSCRIBE命令，客户端可以订阅一个或多个频道，从而成为这些频道的订阅者（subscriber）：每当有其他客户端向被订阅的频道发送消息（message）时，频道的所有订阅者都会收到这条消息。

举个例子，假设A、B、C三个客户端都执行了命令：

```
SUBSCRIBE "news.it"
```

那么这三个客户端就是"news.it"频道的订阅者，如图18-1所示。

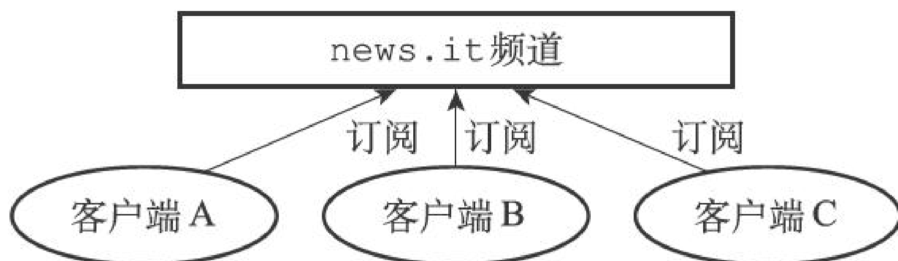


图18-1　news.it频道和它的三个订阅者

如果这时某个客户端执行命令

```
PUBLISH "news.it" "hello"
```

向"news.it"频道发送消息"hello"，那么"news.it"的三个订阅者都将收到这条消息，如图18-2所示。

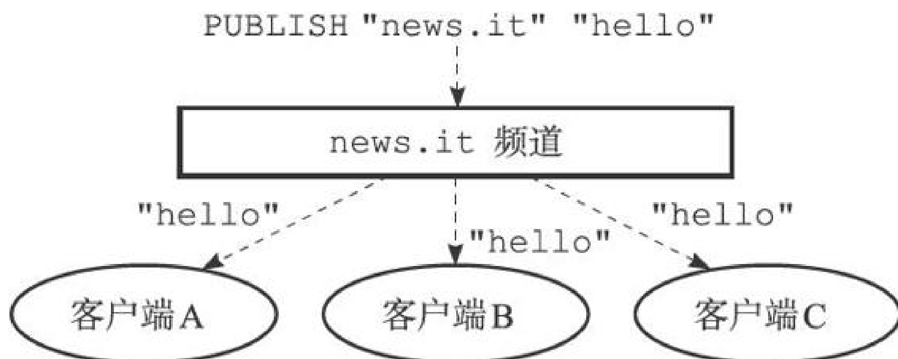


图18-2　向news.it频道发送消息

除了订阅频道之外，客户端还可以通过执行PSUBSCRIBE命令订阅一个或多个模式，从而成为这些模式的订阅者：每当有其他客户端向某个频道发送消息时，消息不仅会被发送给这个频道的所有订阅者，它还会被发送给所有与这个频道相匹配的模式的订阅者。

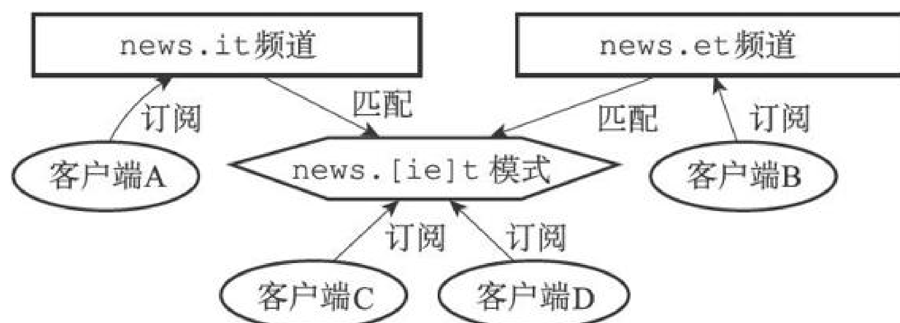


图18-3　频道和模式的订阅状态

举个例子，假设如图18-3所示：

·客户端A正在订阅频道"news.it"。

·客户端B正在订阅频道"news.et"。

·客户端C和客户端D正在订阅与"news.it"频道和"news.et"频道相匹配的模式"news.[ie]t"。

如果这时某个客户端执行命令

```
PUBLISH "news.it" "hello"
```

向"news.it"频道发送消息"hello"，那么不仅正在订阅"news.it"频道的客户端A会收到消息，客户端C和客户端D也同样会收到消息，因为这两个客户端正在订阅匹配"news.it"频道的"news.[ie]t"模式，如图18-4所示。

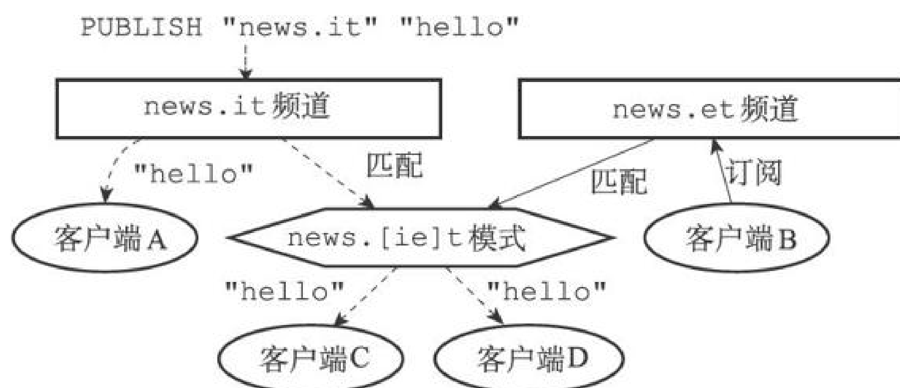


图18-4　将消息发送给频道的订阅者和匹配模式的订阅者（1）

与此类似，如果某个客户端执行命令

```
PUBLISH "news.et" "world"
```

向"news.et"频道发送消息"world"，那么不仅正在订阅"news.et"频道的客户端B会收到消息，客户端C和客户端D也同样会收到消息，因为这两个客户端正在订阅匹配"news.et"频道的"news.[ie]t"模式，如图18-5所示。

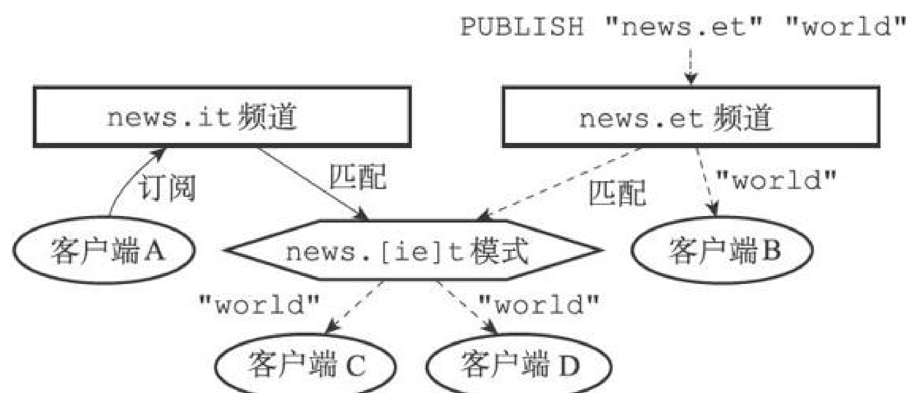


图18-5　将消息发送给频道的订阅者和匹配模式的订阅者（2）

本章接下来的内容将首先介绍订阅频道的SUBSCRIBE命令和退订频道的UNSUBSCRIBE命令的实现原理，然后介绍订阅模式的PSUBSCRIBE命令和退订模式的PUNSUBSCRIBE命令的实现原理。

在介绍完以上四个命令的实现原理之后，本章会对PUBLISH命令的实现原理进行介绍，说明消息是如何发送给频道的订阅者以及模式的订阅者的。

最后，本章将对Redis　2.8新引入的PUBSUB命令的三个子命令进行介绍，并说明这三个子命令的实现原理。

18.1 频道的订阅与退订

当一个客户端执行SUBSCRIBE命令订阅某个或某些频道的时候，这个客户端与被订阅频道之间就建立起了一种订阅关系。

Redis将所有频道的订阅关系都保存在服务器状态的pubsub_channels字典里面，这个字典的键是某个被订阅的频道，而键的值则是一个链表，链表里面记录了所有订阅这个频道的客户端：

```
struct redisServer {
  // ...
  //
保存所有频道的订阅关系
  dict *pubsub_channels;
  // ...
};
```

比如说，图18-6就展示了一个pubsub_channels字典示例，这个字典记录了以下信息：

·client-1、client-2、client-3三个客户端正在订阅"news.it"频道。

·客户端client-4正在订阅"news.sport"频道。

·client-5和client-6两个客户端正在订阅"news.business"频道。

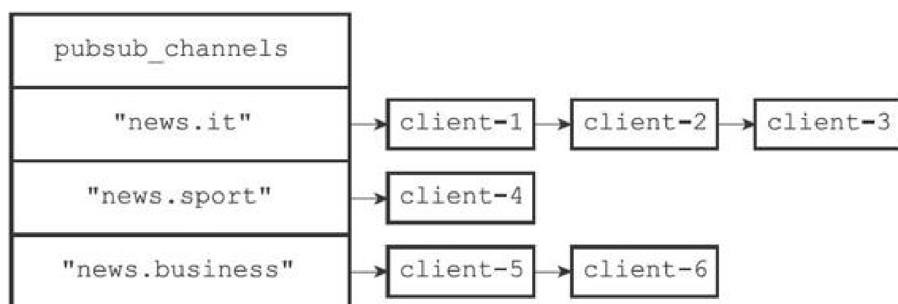


图18-6　一个pubsub_channels字典示例

18.1.1 订阅频道

每当客户端执行SUBSCRIBE命令订阅某个或某些频道的时候，服务器都会将客户端与被订阅的频道在pubsub_channels字典中进行关联。

根据频道是否已经有其他订阅者，关联操作分为两种情况执行：

·如果频道已经有其他订阅者，那么它在pubsub_channels字典中必然有相应的订阅者链表，程序唯一要做的就是将客户端添加到订阅者链表的末尾。

·如果频道还未有任何订阅者，那么它必然不存在于pubsub_channels字典，程序首先要在pubsub_channels字典中为频道创建一个键，并将这个键的值设置为空链表，然后再将客户端添加到链表，成为链表的第一个元素。

举个例子，假设服务器pubsub_channels字典的当前状态如图18-6所示，那么当客户端client-10086执行命令

```
SUBSCRIBE "news.sport" "news.movie"
```

之后，pubsub_channels字典将更新至图18-7所示的状态，其中用虚线包围的是新添加的节点：

·更新后的pubsub_channels字典新增了"news.movie"键，该键对应的链表值只包含一个client-10086节点，表示目前只有client-10086一个客户端在订阅"news.movie"频道。

·至于原本就已经有客户端在订阅的"news.sport"频道，client-10086的节点放在了频道对应链表的末尾，排在client-4节点的后面。

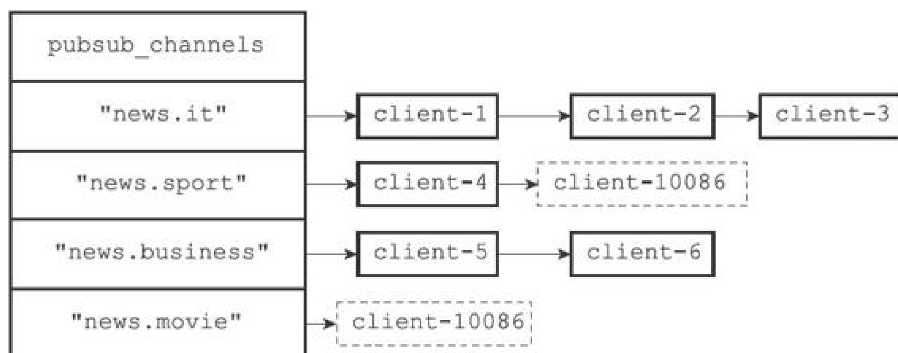


图18-7　执行SUBSCRIBE之后的pubsub_channels字典

SUBSCRIBE命令的实现可以用以下伪代码来描述：

```
def subscribe(*all_input_channels):
  #
遍历输入的所有频道
  for channel in all_input_channels:
    #
如果channel
不存在于pubsub_channels
字典（没有任何订阅者）
    #
那么在字典中添加channel
键，并设置它的值为空链表
    if channel not in server.pubsub_channels:
    server.pubsub_channels[channel] = []
    #
将订阅者添加到频道所对应的链表的末尾
    server.pubsub_channels[channel].append(client)
```

18.1.2 退订频道

UNSUBSCRIBE命令的行为和SUBSCRIBE命令的行为正好相反，当一个客户端退订某个或某些频道的时候，服务器将从pubsub_channels中解除客户端与被退订频道之间的关联：

·程序会根据被退订频道的名字，在pubsub_channels字典中找到频道对应的订阅者链表，然后从订阅者链表中删除退订客户端的信息。

·如果删除退订客户端之后，频道的订阅者链表变成了空链表，那么说明这个频道已经没有任何订阅者了，程序将从pubsub_channels字典中删除频道对应的键。

举个例子，假设pubsub_channels的当前状态如图18-8所示，那么当客户端client-10086执行命令

```
UNSUBSCRIBE "news.sport" "news.movie"
```

之后，图中用虚线包围的两个节点将被删除（如图18-9所示）：

·在pubsub_channels字典更新之后，client-10086的信息已经从"news.sport"频道和"news.movie"频道的订阅者链表中被删除了。

·另外，因为删除client-10086之后，频道"news.movie"已经没有任何订阅者，因此键"news.movie"也从字典中被删除了。

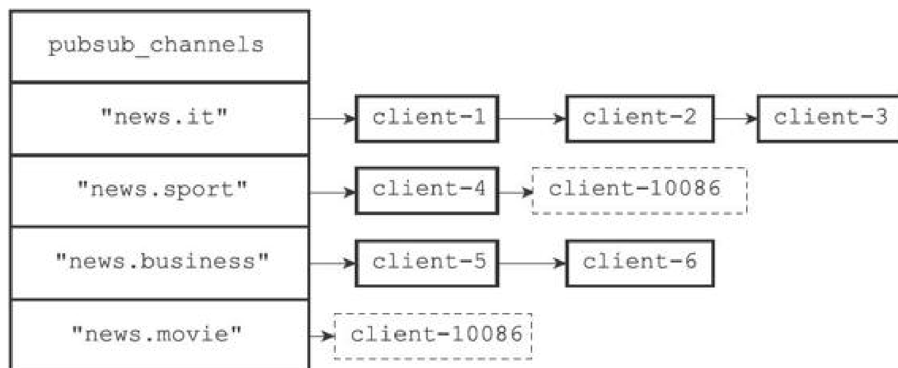


图18-8 执行UNSUBSCRIBE之前的pubsub_channels字典

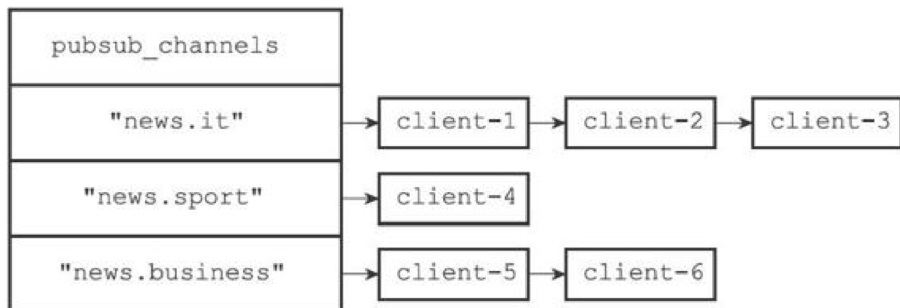


图18-9　执行UNSUBSCRIBE之后的pubsub_channels字典

UNSUBSCRIBE命令的实现可以用以下伪代码来描述：

```
def unsubscribe(*all_input_channels):
  #
遍历要退订的所有频道
  for channel in all_input_channels:
    #
在订阅者链表中删除退订的客户端
    server.pubsub_channels[channel].remove(client)
    #
如果频道已经没有任何订阅者了（订阅者链表为空）
    #
那么将频道从字典中删除
    if len(server.pubsub_channels[channel]) == 0:
    server.pubsub_channels.remove(channel)
```

18.2　模式的订阅与退订

前面说过，服务器将所有频道的订阅关系都保存在服务器状态的pubsub_channels属性里面，与此类似，服务器也将所有模式的订阅关系都保存在服务器状态的pubsub_patterns属性里面：

```
struct redisServer {
  // ...
  //
保存所有模式订阅关系
  list *pubsub_patterns;
  // ...
};
```

pubsub_patterns属性是一个链表，链表中的每个节点都包含着一个pubsub　Pattern结构，这个结构的pattern属性记录了被订阅的模式，而client属性则记录了订阅模式的客户端：

```
typedef struct pubsubPattern {
  //
订阅模式的客户端
  redisClient *client;
  //
被订阅的模式
  robj *pattern;
} pubsubPattern;
```

图18-10是一个pubsubPattern结构示例，它显示客户端client-9正在订阅模式"news.*"。

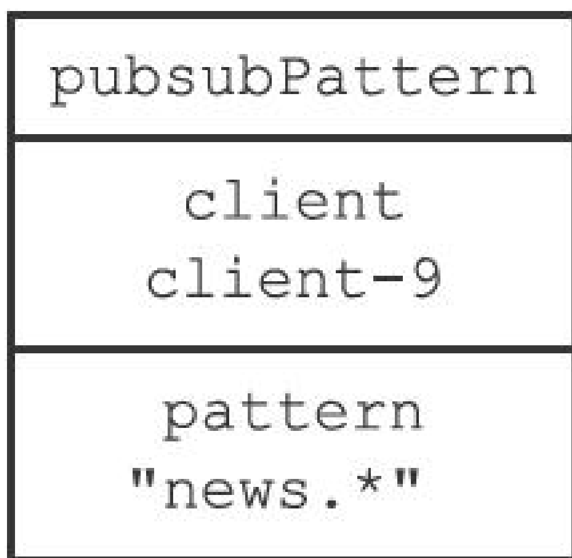


图18-10　pubsubPattern结构示例

图18-11展示了一个pubsub_patterns链表示例，这个链表记录了以

下信息：

·客户端client-7正在订阅模式"music.*"。

·客户端client-8正在订阅模式"book.*"。

·客户端client-9正在订阅模式"news.*"。

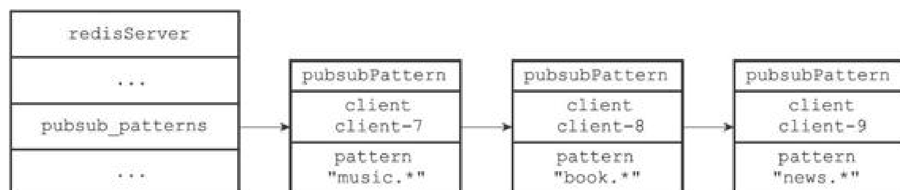


图18-11　pubsub_patterns链表示例

18.2.1　订阅模式

每当客户端执行PSUBSCRIBE命令订阅某个或某些模式的时候，服务器会对每个被订阅的模式执行以下两个操作：

1）新建一个pubsubPattern结构，将结构的pattern属性设置为被订阅的模式，client属性设置为订阅模式的客户端。

2）将pubsubPattern结构添加到pubsub_patterns链表的表尾。举个例子，假设服务器中pubsub_patterns链表的当前状态如图18-12所示。

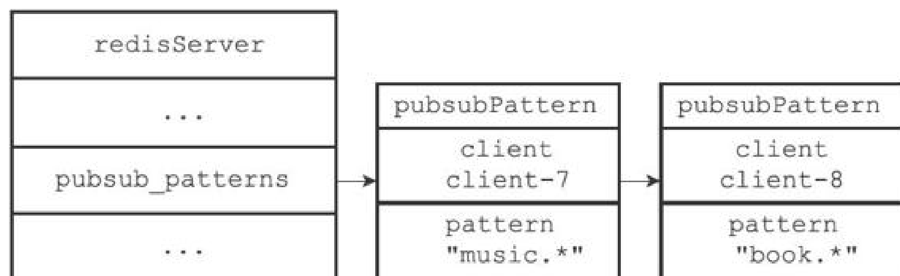


图18-12　执行PSUBSCRIBE命令之前的pubsub_patterns链表

那么当客户端client-9执行命令

```
PSUBSCRIBE "news.*"
```

之后，pubsub_patterns链表将更至新图18-13所示的状态，其中用虚线包围的是新添加的pubsubPattern结构。

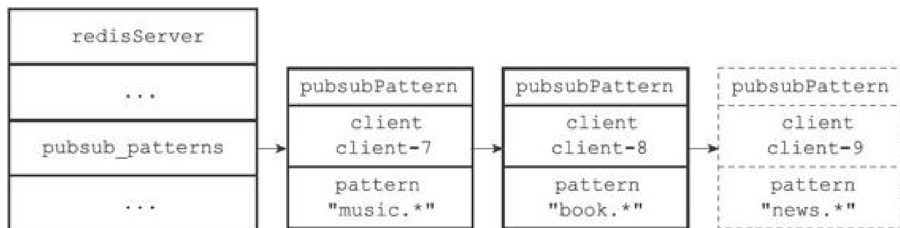


图18-13　执行PSUBSCRIBE命令之后的pubsub_patterns链表

PSUBSCRIBE命令的实现原理可以用以下伪代码来描述：

```
def psubscribe(*all_input_patterns):
    #
遍历输入的所有模式
    for pattern in all_input_patterns:
        #
创建新的pubsubPattern
结构
        #
记录被订阅的模式，以及订阅模式的客户端
        pubsubPattern = create_new_pubsubPattern()
        pubsubPattern.client = client
        pubsubPattern.pattern = pattern
        #
将新的pubsubPattern
追加到pubsub_patterns
链表末尾
        server.pubsub_patterns.append(pubsubPattern)
```

18.2.2　退订模式

模式的退订命令PUNSUBSCRIBE是PSUBSCRIBE命令的反操作：当一个客户端退订某个或某些模式的时候，服务器将在pubsub_patterns链表中查找并删除那些pattern属性为被退订模式，并且client属性为执行退订命令的客户端的pubsubPattern结构。

举个例子，假设服务器pubsub_patterns链表的当前状态如图18-14所示。

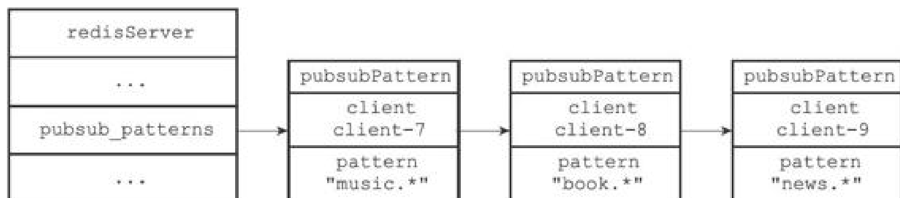


图18-14　执行PUNSUBSCRIBE命令之前的pubsub_patterns链表

那么当客户端client-9执行命令

```
PUNSUBSCRIBE "news.*"
```

之后，client属性为client-9，pattern属性为"news.*"的pubsubPattern结构将被删除，pubsub_patterns链表将更新至图18-15所示的样子。

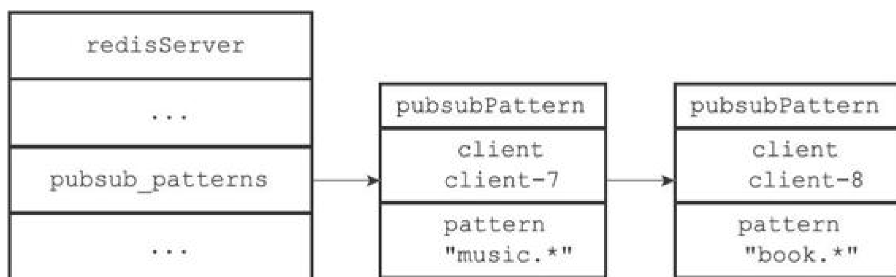


图18-15　执行PUNSUBSCRIBE命令之后的pubsub_patterns链表

PUNSUBSCRIBE命令的实现原理可以用以下伪代码来描述：

```
def punsubscribe(*all_input_patterns):
  #
遍历所有要退订的模式
  for pattern in all_input_patterns:
    #
遍历pubsub_patterns
链表中的所有pubsubPattern
结构
    for pubsubPattern in server.pubsub_patterns:
    #
如果当前客户端和pubsubPattern
记录的客户端相同
    #
并且要退订的模式也和pubsubPattern
记录的模式相同
    if client == pubsubPattern.client and \
     pattern == pubsubPattern.pattern:
     #
那么将这个pubsubPattern
从链表中删除
     server.pubsub_patterns.remove(pubsubPattern)
```

18.3　发送消息

当一个Redis客户端执行PUBLISH <channel> <message>命令将消息message发送给频道channel的时候，服务器需要执行以下两个动作：

1）将消息message发送给channel频道的所有订阅者。

2）如果有一个或多个模式pattern与频道channel相匹配，那么将消息message发送给pattern模式的订阅者。

接下来的两个小节将分别介绍这两个动作的实现方式。

18.3.1　将消息发送给频道订阅者

因为服务器状态中的pubsub_channels字典记录了所有频道的订阅关系，所以为了将消息发送给channel频道的所有订阅者，PUBLISH命令要做的就是在pubsub_channels字典里找到频道channel的订阅者名单（一个链表），然后将消息发送给名单上的所有客户端。举个例子，假设服务器pubsub_channels字典当前的状态如图18-16所示。

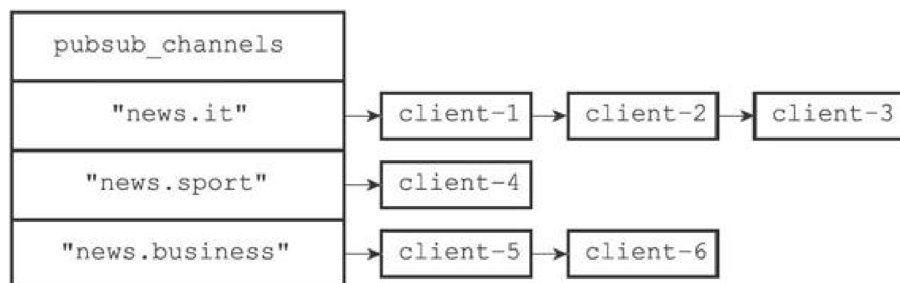


图18-16　pubsub_channels字典

如果这时某个客户端执行命令

```
PUBLISH "news.it" "hello"
```

那么PUBLISH命令将在pubsub_channels字典中查找键"news.it"对应的链表值，并通过遍历链表将消息"hello"发送给"news.it"频道的三个订阅者：client-1、client-2和client-3。

PUBLISH命令将消息发送给频道订阅者的方法可以用以下伪代码来描述：

```
def channel_publish(channel, message):
    #
如果channel
```

```
键不存在于pubsub_channels
字典中
  #
那么说明channel
频道没有任何订阅者
  #
程序不做发送动作，直接返回
  if channel not in server.pubsub_channels:
   return
  #
运行到这里，说明channel
频道至少有一个订阅者
  #
程序遍历channel
频道的订阅者链表
  #
将消息发送给所有订阅者
  for subscriber in server.pubsub_channels[channel]:
    send_message(subscriber, message)
```

18.3.2　将消息发送给模式订阅者

因为服务器状态中的pubsub_patterns链表记录了所有模式的订阅关系，所以为了将消息发送给所有与channel频道相匹配的模式的订阅者，PUBLISH命令要做的就是遍历整个pubsub_patterns链表，查找那些与channel频道相匹配的模式，并将消息发送给订阅了这些模式的客户端。

举个例子，假设pubsub_patterns链表的当前状态如图18-17所示。

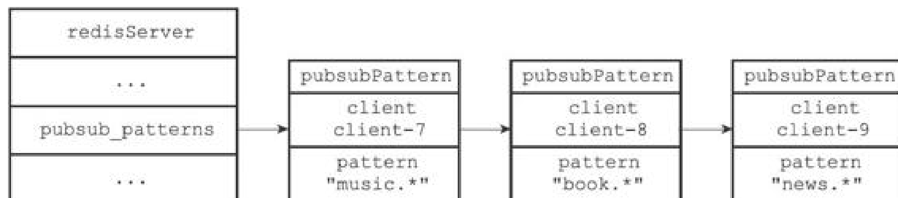


图18-17　pubsub_patterns链表

如果这时某个客户端执行命令

```
PUBLISH "news.it" "hello"
```

那么PUBLISH命令会首先将消息"hello"发送给"news.it"频道的所有订阅者，然后开始在pubsub_patterns链表中查找是否有被订阅的模式与"news.it"频道相匹配，结果发现"news.it"频道和客户端client-9订阅的"news.*"频道匹配，于是命令将消息"hello"发送给客户端client-9。

PUBLISH命令将消息发送给模式订阅者的方法可以用以下伪代码来描述：

```
def pattern_publish(channel, message):
  #
遍历所有模式订阅消息
```

```
  for pubsubPattern in server.pubsub_patterns:
    #
如果频道和模式相匹配
    if match(channel, pubsubPattern.pattern):
          #
那么将消息发送给订阅该模式的客户端
          send_message(pubsubPattern.client, message)
```

最后，PUBLISH命令的实现可以用以下伪代码来描述：

```
def publish(channel, message):
  #
将消息发送给channel
频道的所有订阅者
  channel_publish(channel, message)
  #
将消息发送给所有和channel
频道相匹配的模式的订阅者
  pattern_publish(channel, message)
```

18.4 查看订阅信息

PUBSUB命令是Redis 2.8新增加的命令之一，客户端可以通过这个命令来查看频道或者模式的相关信息，比如某个频道目前有多少订阅者，又或者某个模式目前有多少订阅者，诸如此类。

以下三个小节将分别介绍PUBSUB命令的三个子命令，以及这些子命令的实现原理。

18.4.1 PUBSUB CHANNELS

PUBSUB CHANNELS[pattern]子命令用于返回服务器当前被订阅的频道，其中pattern参数是可选的：

·如果不给定pattern参数，那么命令返回服务器当前被订阅的所有频道。

·如果给定pattern参数，那么命令返回服务器当前被订阅的频道中那些与pattern模式相匹配的频道。

这个子命令是通过遍历服务器pubsub_channels字典的所有键（每个键都是一个被订阅的频道），然后记录并返回所有符合条件的频道来实现的，这个过程可以用以下伪代码来描述：

```
def pubsub_channels(pattern=None):
  #
一个列表，用于记录所有符合条件的频道
  channel_list = []
  #
遍历服务器中的所有频道
  #
（也即是pubsub_channels
字典的所有键）
  for channel in server.pubsub_channels:
    #
当以下两个条件的任意一个满足时，将频道添加到链表里面：
    #1
）用户没有指定pattern
参数
    #2
）用户指定了pattern
参数，并且channel
和pattern
匹配
    if (pattern is None) or match(channel, pattern):
     channel_list.append(channel)
#
向客户端返回频道列表
return channel_list
```

举个例子，对于图18-18所示的pubsub_channels字典来说，执行PUBSUB CHANNELS命令将返回服务器目前被订阅的四个频道：

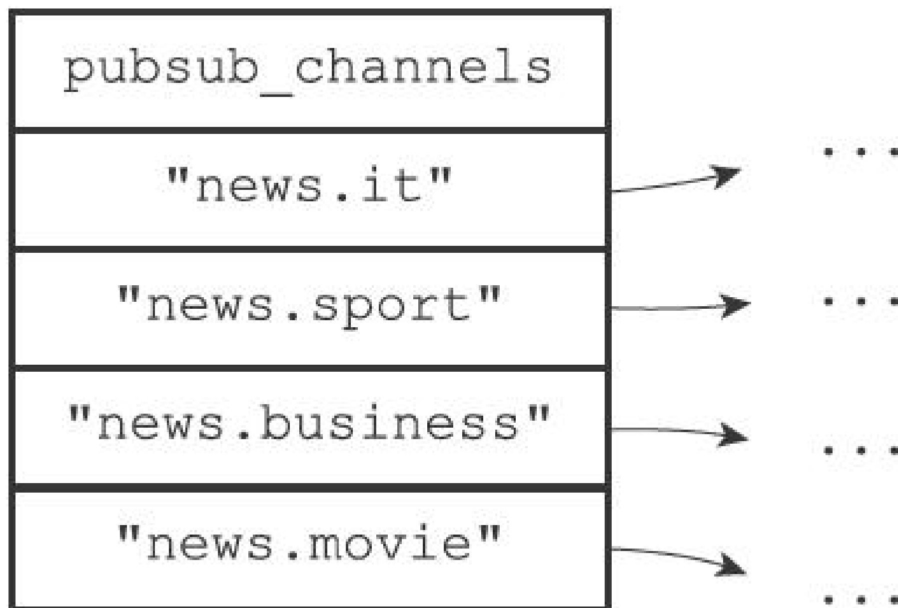


图18-18　pubsub_channels字典示例

```
redis> PUBSUB CHANNELS
1) "news.it"
2) "news.sport"
3) "news.business"
4) "news.movie"
```

另一方面，执行PUBSUB　　　　CHANNELS"news.[is]*"命令将返回"news.it"和"news.sport"两个频道，因为只有这两个频道和"news.[is]*"模式相匹配：

```
redis> PUBSUB CHANNELS "news.[is]*"
1) "news.it"
2) "news.sport"
```

18.4.2　PUBSUB NUMSUB

PUBSUB NUMSUB[channel-1 channel-2...channel-n]子命令接受任意多个频道作为输入参数，并返回这些频道的订阅者数量。

这个子命令是通过在pubsub_channels字典中找到频道对应的订阅者链表，然后返回订阅者链表的长度来实现的（订阅者链表的长度就是频道订阅者的数量），这个过程可以用以下伪代码来描述：

```
def pubsub_numsub(*all_input_channels):
  #
遍历输入的所有频道
  for channel in all_input_channels:
       #
如果pubsub_channels
```

```
字典中没有channel
这个键
          #
那么说明channel
频道没有任何订阅者
        if channel not in server.pubsub_channels:
            #
返回频道名
            reply_channel_name(channel)
            #
订阅者数量为0
            reply_subscribe_count(0)
   #
如果pubsub_channels
字典中存在channel
键
   #
那么说明channel
频道至少有一个订阅者
   else:
      #
返回频道名
      reply_channel_name(channel)
      #
订阅者链表的长度就是订阅者数量
      reply_subscribe_count(len(server.pubsub_channels[channel]))
```

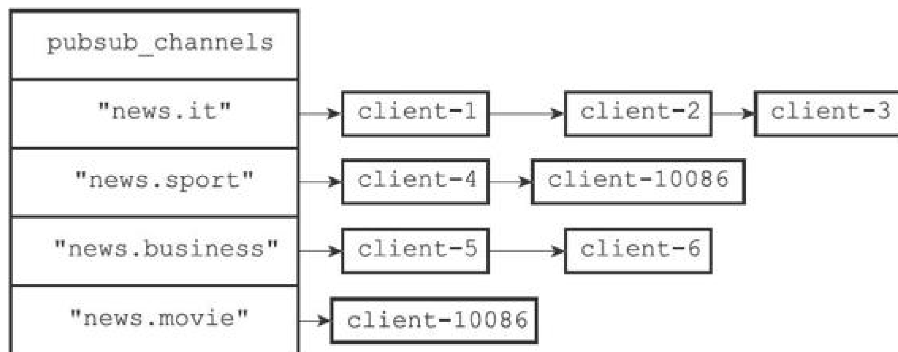


图18-19　pubsub_channels字典

举个例子，对于图18-19所示的pubsub_channels字典来说，对字典中的四个频道执行PUBSUB NUMSUB命令将获得以下回复：

```
redis> PUBSUB NUMSUB news.it news.sport news.business news.movie
1) "news.it"
2) "3"
3) "news.sport"
4) "2"
5) "news.business"
6) "2"
7) "news.movie"
8) "1"
```

18.4.3　PUBSUB NUMPAT

PUBSUB NUMPAT子命令用于返回服务器当前被订阅模式的数量。

这个子命令是通过返回pubsub_patterns链表的长度来实现的，因为

这个链表的长度就是服务器被订阅模式的数量，这个过程可以用以下伪代码来描述：

```
def pubsub_numpat():
  # pubsub_patterns
链表的长度就是被订阅模式的数量
  reply_pattern_count(len(server.pubsub_patterns))
```

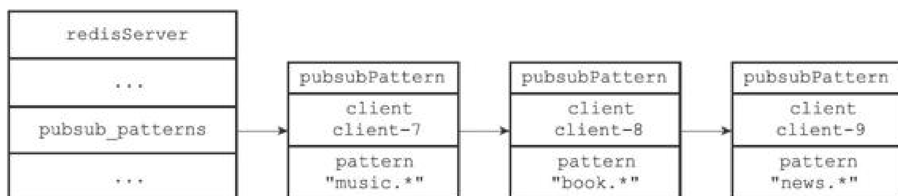


图18-20　pubsub_patterns链表

举个例子，对于图18-20所示的pubsub_patterns链表来说，执行PUBSUB NUMPAT命令将返回3：

```
redis> PUBSUB NUMPAT
(integer) 3
```

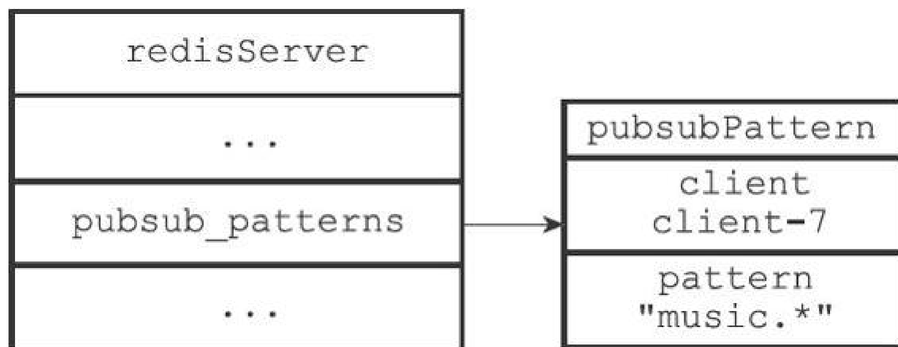


图18-21　pubsub_patterns链表

而对于图18-21所示的pubsub_patterns链表来说，执行PUBSUB NUMPAT命令将返回1：

```
redis> PUBSUB NUMPAT
(integer) 1
```

18.5 重点回顾

·服务器状态在pubsub_channels字典保存了所有频道的订阅关系：SUBSCRIBE命令负责将客户端和被订阅的频道关联到这个字典里面，而UNSUBSCRIBE命令则负责解除客户端和被退订频道之间的关联。

·服务器状态在pubsub_patterns链表保存了所有模式的订阅关系：PSUBSCRIBE命令负责将客户端和被订阅的模式记录到这个链表中，而PUNSUBSCRIBE命令则负责移除客户端和被退订模式在链表中的记录。

·PUBLISH命令通过访问pubsub_channels字典来向频道的所有订阅者发送消息，通过访问pubsub_patterns链表来向所有匹配频道的模式的订阅者发送消息。

·PUBSUB命令的三个子命令都是通过读取pubsub_channels字典和pubsub_patterns链表中的信息来实现的。

18.6 参考资料

·关于发布与订阅模式的定义可以参考维基百科的Publish Subscribe Pattern词条：http://en.wikipedia.org/wiki/Publish-subscribe_pattern，以及《设计模式》一书的5.7节。

·《Pattern-Oriented Software Architecture Volume 4，A Pattern Language for Distributed Computing》一书第10章《Distribution Infrastructure》关于信息、信息传递、发布与订阅等主题的讨论非常好，值得一看。

·维基百科的Glob词条给出了Glob风格模式匹配的简介：http://en.wikipedia.org/wiki/Glob_（programming），具体的匹配符语法可以参考glob（7）手册的Wildcard Matching小节。

第19章　事务

Redis通过MULTI、EXEC、WATCH等命令来实现事务（transaction）功能。事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求。

以下是一个事务执行的过程，该事务首先以一个MULTI命令为开始，接着将多个命令放入事务当中，最后由EXEC命令将这个事务提交（commit）给服务器执行：

```
redis> MULTI
OK
redis> SET "name" "Practical Common Lisp"
QUEUED
redis> GET "name"
QUEUED
redis> SET "author" "Peter Seibel"
QUEUED
redis> GET "author"
QUEUED
redis> EXEC
1) OK
2) "Practical Common Lisp"
3) OK
4) "Peter Seibel"
```

在本章接下来的内容中，我们首先会介绍Redis如何使用MULTI和EXEC命令来实现事务功能，说明事务中的多个命令是如何被保存到事务里面的，而这些命令又是如何被执行的。

在介绍了事务的实现原理之后，我们将对WATCH命令的作用进行介绍，并说明WATCH命令的实现原理。

因为事务的安全性和可靠性也是大家关注的焦点，所以本章最后将以常见的ACID性质对Redis事务的原子性、一致性、隔离性和耐久性进行说明。

19.1　事务的实现

一个事务从开始到结束通常会经历以下三个阶段：

1）事务开始。

2）命令入队。

3）事务执行。

本节接下来的内容将对这三个阶段进行介绍，说明一个事务从开始到结束的整个过程。

19.1.1　事务开始

MULTI命令的执行标志着事务的开始：

```
redis> MULTI
OK
```

MULTI命令可以将执行该命令的客户端从非事务状态切换至事务状态，这一切换是通过在客户端状态的flags属性中打开REDIS_MULTI标识来完成的，MULTI命令的实现可以用以下伪代码来表示：

```
def MULTI():
  #
打开事务标识
  client.flags |= REDIS_MULTI
  #
返回OK
回复
  replyOK()
```

19.1.2　命令入队

当一个客户端处于非事务状态时，这个客户端发送的命令会立即被服务器执行：

```
redis> SET "name" "Practical Common Lisp"
OK
redis> GET "name"
"Practical Common Lisp"
redis> SET "author" "Peter Seibel"
OK
redis> GET "author"
"Peter Seibel"
```

与此不同的是，当一个客户端切换到事务状态之后，服务器会根据这个客户端发来的不同命令执行不同的操作：

·如果客户端发送的命令为EXEC、DISCARD、WATCH、MULTI四个命令的其中一个，那么服务器立即执行这个命令。

·与此相反，如果客户端发送的命令是EXEC、DISCARD、WATCH、MULTI四个命令以外的其他命令，那么服务器并不立即执行这个命令，而是将这个命令放入一个事务队列里面，然后向客户端返回QUEUED回复。

服务器判断命令是该入队还是该立即执行的过程可以用流程图19-1来描述。

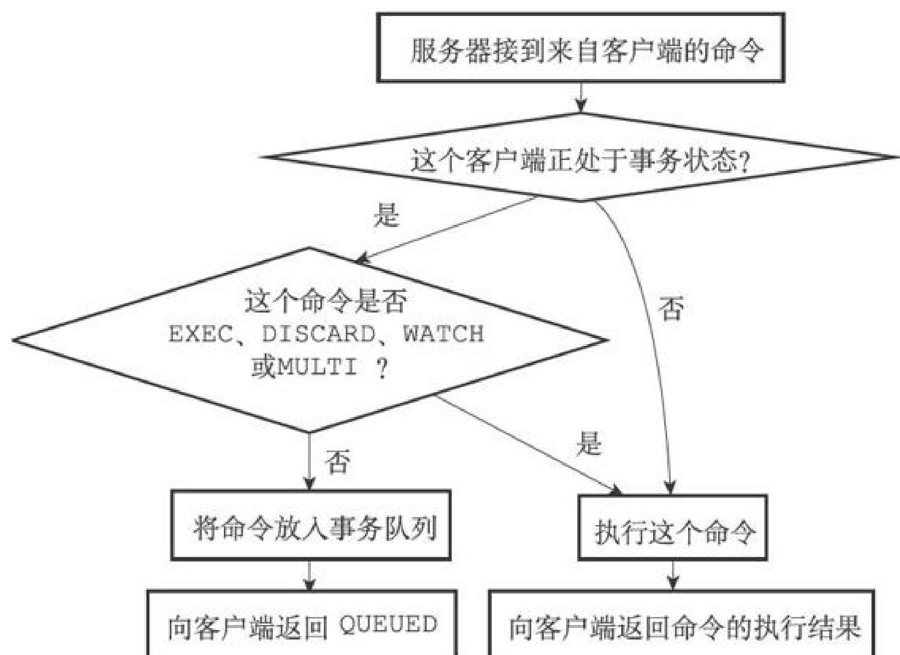


图19-1　服务器判断命令是该入队还是该执行的过程

19.1.3　事务队列

每个Redis客户端都有自己的事务状态，这个事务状态保存在客户端状态的mstate属性里面：

```
typedef struct redisClient {
  // ...
  //
事务状态
  multiState mstate;     /* MULTI/EXEC state */
  // ...
} redisClient;
```

事务状态包含一个事务队列，以及一个已入队命令的计数器（也可以说是事务队列的长度）：

```
typedef struct multiState {
  //
事务队列，FIFO
顺序
  multiCmd *commands;
  //
已入队命令计数
  int count;
} multiState;
```

事务队列是一个multiCmd类型的数组，数组中的每个multiCmd结构都保存了一个已入队命令的相关信息，包括指向命令实现函数的指针、命令的参数，以及参数的数量：

```
typedef struct multiCmd {
  //
参数
  robj **argv;
  //
参数数量
  int argc;
  //
命令指针
  struct redisCommand *cmd;
} multiCmd;
```

事务队列以先进先出（FIFO）的方式保存入队的命令，较先入队的命令会被放到数组的前面，而较后入队的命令则会被放到数组的后面。

举个例子，如果客户端执行以下命令：

```
redis> MULTI
OK
redis> SET "name" "Practical Common Lisp"
QUEUED
redis> GET "name"
QUEUED
redis> SET "author" "Peter Seibel"
QUEUED
redis> GET "author"
QUEUED
```

那么服务器将为客户端创建图19-2所示的事务状态：

·最先入队的SET命令被放在了事务队列的索引0位置上。

·第二入队的GET命令被放在了事务队列的索引1位置上。

·第三入队的另一个SET命令被放在了事务队列的索引2位置上。

·最后入队的另一个GET命令被放在了事务队列的索引3位置上。

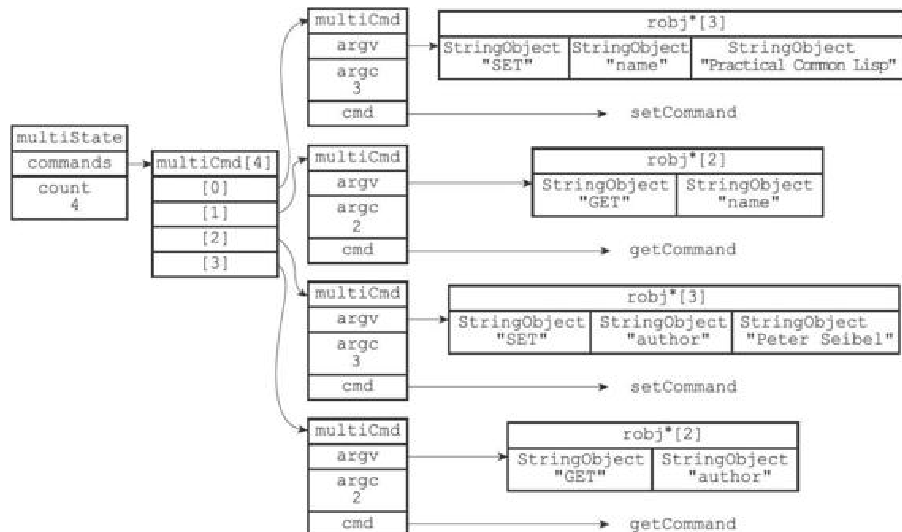


图19-2　事务状态

19.1.4　执行事务

当一个处于事务状态的客户端向服务器发送EXEC命令时，这个EXEC命令将立即被服务器执行。服务器会遍历这个客户端的事务队列，执行队列中保存的所有命令，最后将执行命令所得的结果全部返回给客户端。

举个例子，对于图19-2所示的事务队列来说，服务器首先会执行命令：

```
SET "name" "Practical Common Lisp"
```

接着执行命令：

```
GET "name"
```

之后执行命令：

```
SET "author" "Peter Seibel"
```

再之后执行命令：

```
GET "author"
```

最后，服务器会将执行这四个命令所得的回复返回给客户端：

```
redis> EXEC
1) OK
```

```
2) "Practical Common Lisp"
3) OK
4) "Peter Seibel"
```

EXEC命令的实现原理可以用以下伪代码来描述：

```
def EXEC():
    #
创建空白的回复队列
    reply_queue = []
    #
遍历事务队列中的每个项
    #
读取命令的参数，参数的个数，以及要执行的命令
    for argv, argc, cmd in client.mstate.commands:
      #
执行命令，并取得命令的返回值
      reply = execute_command(cmd, argv, argc)
      #
将返回值追加到回复队列末尾
      reply_queue.append(reply)
    #
移除REDIS_MULTI
标识，让客户端回到非事务状态
    client.flags & = ~REDIS_MULTI
    #
清空客户端的事务状态，包括：
    #1
）清零入队命令计数器
    #2
）释放事务队列
    client.mstate.count = 0
    release_transaction_queue(client.mstate.commands)
    #
将事务的执行结果返回给客户端
    send_reply_to_client(client, reply_queue)
```

19.2 WATCH命令的实现

WATCH命令是一个乐观锁（optimistic locking），它可以在EXEC命令执行之前，监视任意数量的数据库键，并在EXEC命令执行时，检查被监视的键是否至少有一个已经被修改过了，如果是的话，服务器将拒绝执行事务，并向客户端返回代表事务执行失败的空回复。

以下是一个事务执行失败的例子：

```
redis> WATCH "name"
OK
redis> MULTI
OK
redis> SET "name" "peter"
QUEUED
redis> EXEC
(nil)
```

表19-1展示了上面的例子是如何失败的。

表19-1　两个客户端执行命令的过程

时间	客户端 A	客户端 B
T1	WATCH "name"	
T2	MULTI	
T3	SET "name" "peter"	
T4		SET "name" "john"
T5	EXEC	

在时间T4，客户端B修改了"name"键的值，当客户端A在T5执行EXEC命令时，服务器会发现WATCH监视的键"name"已经被修改，因此服务器拒绝执行客户端A的事务，并向客户端A返回空回复。

本节接下来的内容将介绍WATCH命令的实现原理，说明事务系统是如何监视某个键，并在键被修改的情况下，确保事务的安全性的。

19.2.1 使用WATCH命令监视数据库键

每个Redis数据库都保存着一个watched_keys字典，这个字典的键是某个被WATCH命令监视的数据库键，而字典的值则是一个链表，链表中记录了所有监视相应数据库键的客户端：

```
typedef struct redisDb {
  // ...
  //
正在被WATCH
命令监视的键
  dict *watched_keys;
  // ...
```

```
} redisDb;
```

通过watched_keys字典，服务器可以清楚地知道哪些数据库键正在被监视，以及哪些客户端正在监视这些数据库键。

图19-3是一个watched_keys字典的示例，从这个watched_keys字典中可以看出：

·客户端c1和c2正在监视键"name"。

·客户端c3正在监视键"age"。

·客户端c2和c4正在监视键"address"。

通过执行WATCH命令，客户端可以在watched_keys字典中与被监视的键进行关联。举个例子，如果当前客户端为c10086，那么客户端执行以下WATCH命令之后：

```
redis> WATCH "name" "age"
OK
```

图19-3展示的watched_keys字典将被更新至图19-4所示的状态，其中用虚线包围的两个c10086节点就是由刚刚执行的WATCH命令添加到字典中的。

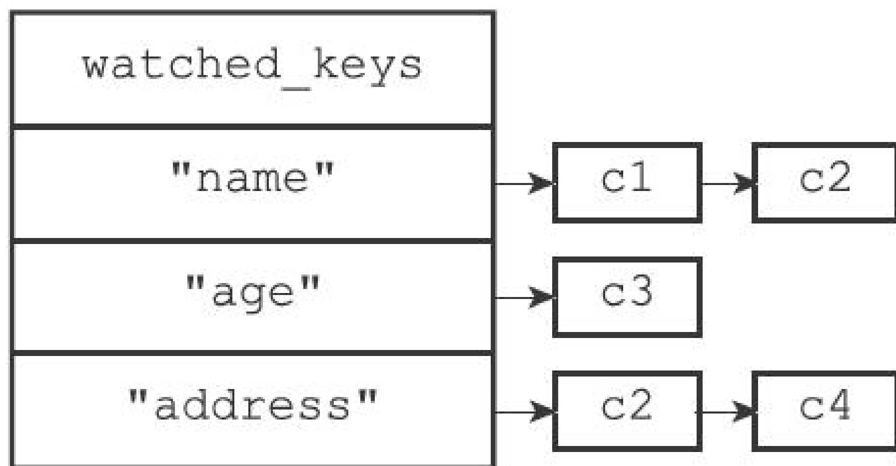


图19-3　一个watched_keys字典

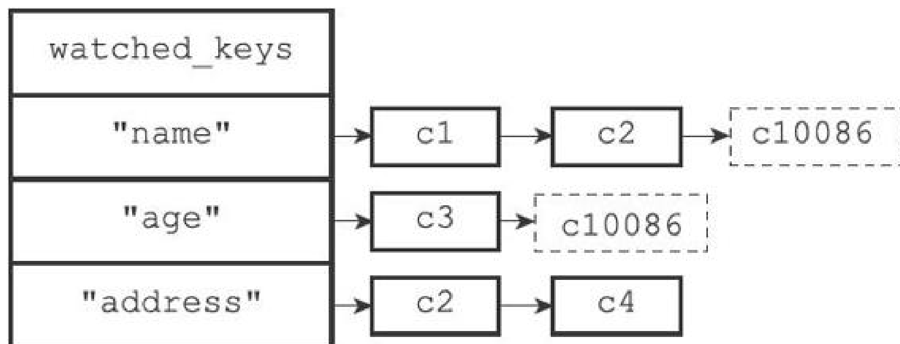


图19-4　执行WATCH命令之后的watched_keys字典

19.2.2　监视机制的触发

所有对数据库进行修改的命令，比如SET、LPUSH、SADD、ZREM、DEL、FLUSHDB等等，在执行之后都会调用multi.c/touchWatchKey函数对watched_keys字典进行检查，查看是否有客户端正在监视刚刚被命令修改过的数据库键，如果有的话，那么touchWatchKey函数会将监视被修改键的客户端的REDIS_DIRTY_CAS标识打开，表示该客户端的事务安全性已经被破坏。

touchWatchKey函数的定义可以用以下伪代码来描述：

```
def touchWatchKey(db, key):
    #
如果键key
存在于数据库的watched_keys
字典中
    #
那么说明至少有一个客户端在监视这个key
    if key in db.watched_keys:
        #
遍历所有监视键key
的客户端
        for client in db.watched_keys[key]:
            #
打开标识
            client.flags |= REDIS_DIRTY_CAS
```

举个例子，对于图19-5所示的watched_keys字典来说：

·如果键"name"被修改，那么c1、c2、c10086三个客户端的REDIS_DIRTY_CAS标识将被打开。

·如果键"age"被修改，那么c3和c10086两个客户端的REDIS_DIRTY_CAS标识将被打开。

·如果键"address"被修改，那么c2和c4两个客户端的REDIS_DIRTY_CAS标识将被打开。

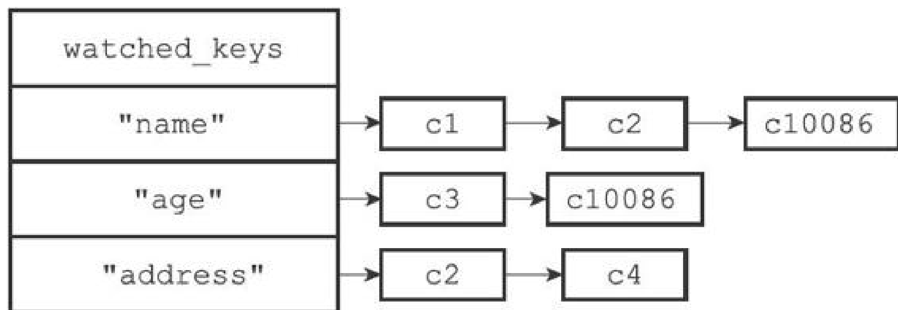


图19-5　watched_keys字典

19.2.3　判断事务是否安全

当服务器接收到一个客户端发来的EXEC命令时，服务器会根据这个客户端是否打开了REDIS_DIRTY_CAS标识来决定是否执行事务：

·如果客户端的REDIS_DIRTY_CAS标识已经被打开，那么说明客户端所监视的键当中，至少有一个键已经被修改过了，在这种情况下，客户端提交的事务已经不再安全，所以服务器会拒绝执行客户端提交的事务。

·如果客户端的REDIS_DIRTY_CAS标识没有被打开，那么说明客户端监视的所有键都没有被修改过（或者客户端没有监视任何键），事务仍然是安全的，服务器将执行客户端提交的这个事务。

这个判断是否执行事务的过程可以用流程图19-6来描述。

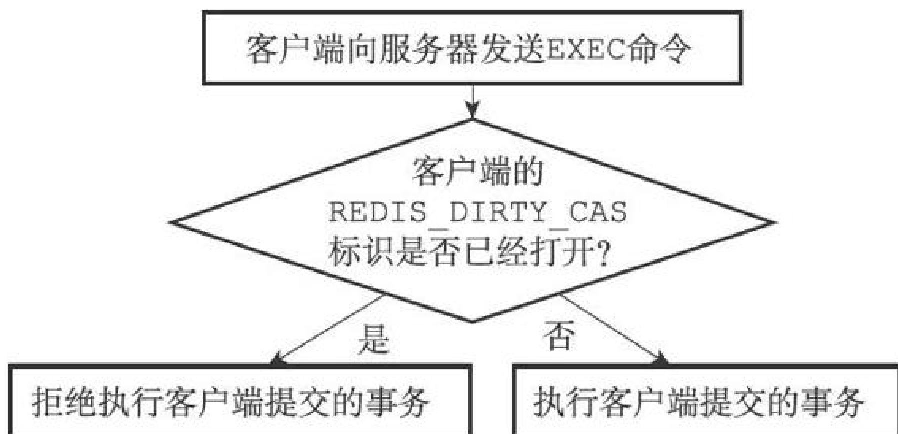


图19-6　服务器判断是否执行事务的过程

举个例子，对于图19-5所示的watched_keys字典来说，如果某个客户端对"name"键进行了修改（比如执行SET"name""john"），那么c1、c2、c10086三个客户端的REDIS_DIRTY_CAS标识将被打开。当这三个客户端向服务器发送EXEC命令的时候，服务器会拒绝执行它们提交的事

务，以此来保证事务的安全性。

19.2.4　一个完整的WATCH事务执行过程

为了进一步熟悉WATCH命令的运作方式，让我们来看一个带有WATCH的事务从开始到失败的整个过程。

假设当前客户端为c10086，而数据库watched_keys字典的当前状态如图19-7所示，那么当c10086执行以下WATCH命令之后：

```
c10086> WATCH "name"
OK
```

watched_keys字典将更新至图19-8所示的状态。

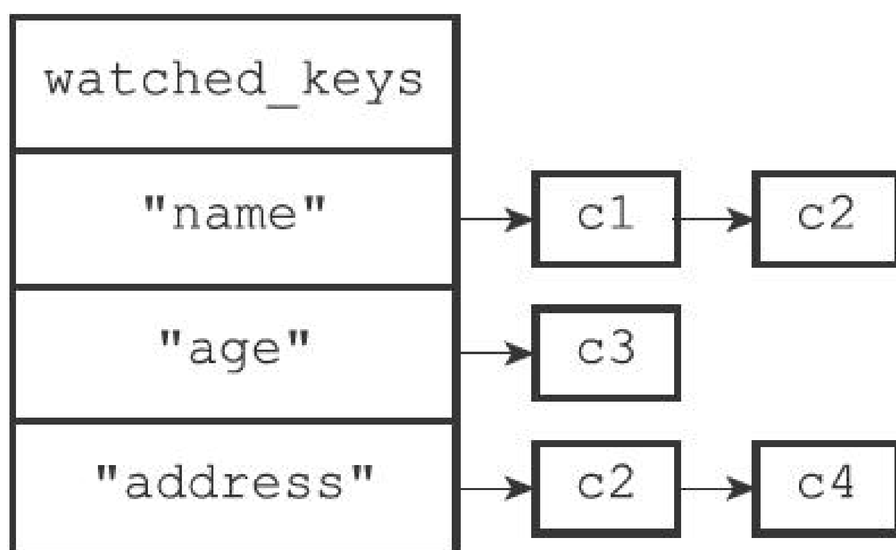


图19-7　执行WATCH命令之前的watched_keys字典

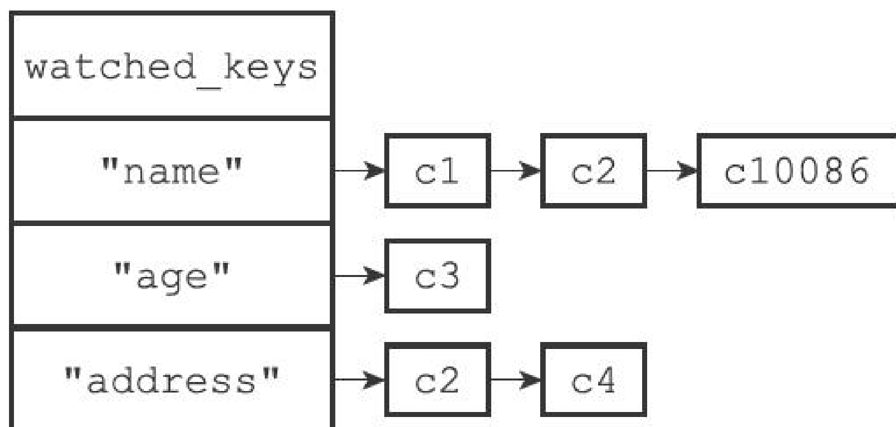


图19-8　执行WATCH命令之后的watched_keys字典

接下来，客户端c10086继续向服务器发送MULTI命令，并将一个SET命令放入事务队列：

```
c10086> MULTI
OK
c10086> SET "name" "peter"
QUEUED
```

就在这时，另一个客户端c999向服务器发送了一条SET命令，将"name"键的值设置成了"john"：

```
c999> SET "name" "john"
OK
```

c999执行的这个SET命令会导致正在监视"name"的所有客户端的REDIS_DIRTY_CAS标识被打开，其中包括客户端c10086。

之后，当c10086向服务器发送EXEC命令时候，因为c10086的REDIS_DIRTY_CAS标志已经被打开，所以服务器将拒绝执行它提交的事务：

```
c10086> EXEC
(nil)
```

19.3　事务的ACID性质

在传统的关系式数据库中，常常用ACID性质来检验事务功能的可靠性和安全性。

在Redis中，事务总是具有原子性（Atomicity）、一致性（Consistency）和隔离性（Isolation），并且当Redis运行在某种特定的持久化模式下时，事务也具有耐久性（Durability）。

以下四个小节将分别对这四个性质进行讨论。

19.3.1　原子性

事务具有原子性指的是，数据库将事务中的多个操作当作一个整体来执行，服务器要么就执行事务中的所有操作，要么就一个操作也不执行。

对于Redis的事务功能来说，事务队列中的命令要么就全部都执行，要么就一个都不执行，因此，Redis的事务是具有原子性的。

举个例子，以下展示的是一个成功执行的事务，事务中的所有命令都会被执行：

```
redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> GET msg
QUEUED
redis> EXEC
1) OK
2) "hello"
```

与此相反，以下展示了一个执行失败的事务，这个事务因为命令入队出错而被服务器拒绝执行，事务中的所有命令都不会被执行：

```
redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> GET
(error) ERR wrong number of arguments for 'get' command
redis> GET msg
QUEUED
redis> EXEC
(error) EXECABORT Transaction discarded because of previous errors.
```

Redis的事务和传统的关系型数据库事务的最大区别在于，Redis不支持事务回滚机制（rollback），即使事务队列中的某个命令在执行期间出现了错误，整个事务也会继续执行下去，直到将事务队列中的所有

命令都执行完毕为止。

在下面的这个例子中，即使RPUSH命令在执行期间出现了错误，事务的后续命令也会继续执行下去，并且之前执行的命令也不会有任何影响：

```
redis> SET msg "hello" # msg
键是一个字符串
OK
redis> MULTI
OK
redis> SADD fruit "apple" "banana" "cherry"
QUEUED
redis> RPUSH msg "good bye" "bye bye" #
错误地对字符串键msg
执行列表键的命令
QUEUED
redis> SADD alphabet "a" "b" "c"
QUEUED
redis> EXEC
1) (integer) 3
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
3) (integer) 3
```

Redis的作者在事务功能的文档中解释说，不支持事务回滚是因为这种复杂的功能和Redis追求简单高效的设计主旨不相符，并且他认为，Redis事务的执行时错误通常都是编程错误产生的，这种错误通常只会出现在开发环境中，而很少会在实际的生产环境中出现，所以他认为没有必要为Redis开发事务回滚功能。

19.3.2　一致性

事务具有一致性指的是，如果数据库在执行事务之前是一致的，那么在事务执行之后，无论事务是否执行成功，数据库也应该仍然是一致的。

“一致”指的是数据符合数据库本身的定义和要求，没有包含非法或者无效的错误数据。

Redis通过谨慎的错误检测和简单的设计来保证事务的一致性，以下三个小节将分别介绍三个Redis事务可能出错的地方，并说明Redis是如何妥善地处理这些错误，从而确保事务的一致性的。

1.入队错误

如果一个事务在入队命令的过程中，出现了命令不存在，或者命令的格式不正确等情况，那么Redis将拒绝执行这个事务。

在以下展示的示例中，因为客户端尝试向事务入队一个不存在的命令YAHOOOO，所以客户端提交的事务会被服务器拒绝执行：

```
redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> YAHOOOO
(error) ERR unknown command 'YAHOOOO'
redis> GET msg
QUEUED
redis> EXEC
(error) EXECABORT Transaction discarded because of previous errors.
```

因为服务器会拒绝执行入队过程中出现错误的事务，所以Redis事务的一致性不会被带有入队错误的事务影响。

Redis 2.6.5以前的入队错误处理

根据文档记录，在Redis 2.6.5以前的版本，即使有命令在入队过程中发生了错误，事务一样可以执行，不过被执行的命令只包括那些正确入队的命令。以下这段代码是在Redis 2.6.4版本上测试的，可以看到，事务可以正常执行，但只有成功入队的SET命令和GET命令被执行了，而错误的YAHOOOO则被忽略了：

```
redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> YAHOOOO
(error) ERR unknown command 'YAHOOOO'
redis> GET msg
QUEUED
redis> EXEC
1) OK
2) "hello"
```

因为错误的命令不会被入队，所以Redis不会尝试去执行错误的命令，因此，即使在2.6.5以前的版本中，Redis事务的一致性也不会被入队错误影响。

2.执行错误

除了入队时可能发生错误以外，事务还可能在执行的过程中发生错误。

关于这种错误有两个需要说明的地方：

·执行过程中发生的错误都是一些不能在入队时被服务器发现的错误，这些错误只会在命令实际执行时被触发。

·即使在事务的执行过程中发生了错误，服务器也不会中断事务的执

行，它会继续执行事务中余下的其他命令，并且已执行的命令（包括执行命令所产生的结果）不会被出错的命令影响。

对数据库键执行了错误类型的操作是事务执行期间最常见的错误之一。

在下面展示的这个例子中，我们首先用SET命令将键"msg"设置成了一个字符串键，然后在事务里面尝试对"msg"键执行只能用于列表键的RPUSH命令，这将引发一个错误，并且这种错误只能在事务执行（也即是命令执行）期间被发现：

```
redis> SET msg "hello"
OK
redis> MULTI
OK
redis> SADD fruit "apple" "banana" "cherry"
QUEUED
redis> RPUSH msg "good bye" "bye bye"
QUEUED
redis> SADD alphabet "a" "b" "c"
QUEUED
redis> EXEC
1) (integer) 3
2) (error) WRONGTYPE Operation against a key holding the wrong kind of value
3) (integer) 3
```

因为在事务执行的过程中，出错的命令会被服务器识别出来，并进行相应的错误处理，所以这些出错命令不会对数据库做任何修改，也不会对事务的一致性产生任何影响。

3.服务器停机

如果Redis服务器在执行事务的过程中停机，那么根据服务器所使用的持久化模式，可能有以下情况出现：

·如果服务器运行在无持久化的内存模式下，那么重启之后的数据库将是空白的，因此数据总是一致的。

·如果服务器运行在RDB模式下，那么在事务中途停机不会导致不一致性，因为服务器可以根据现有的RDB文件来恢复数据，从而将数据库还原到一个一致的状态。如果找不到可供使用的RDB文件，那么重启之后的数据库将是空白的，而空白数据库总是一致的。

·如果服务器运行在AOF模式下，那么在事务中途停机不会导致不一致性，因为服务器可以根据现有的AOF文件来恢复数据，从而将数据库还原到一个一致的状态。如果找不到可供使用的AOF文件，那么重启之后的数据库将是空白的，而空白数据库总是一致的。

综上所述，无论Redis服务器运行在哪种持久化模式下，事务执行中途发生的停机都不会影响数据库的一致性。

19.3.3 隔离性

事务的隔离性指的是，即使数据库中有多个事务并发地执行，各个事务之间也不会互相影响，并且在并发状态下执行的事务和串行执行的事务产生的结果完全相同。

因为Redis使用单线程的方式来执行事务（以及事务队列中的命令），并且服务器保证，在执行事务期间不会对事务进行中断，因此，Redis的事务总是以串行的方式运行的，并且事务也总是具有隔离性的。

19.3.4 耐久性

事务的耐久性指的是，当一个事务执行完毕时，执行这个事务所得的结果已经被保存到永久性存储介质（比如硬盘）里面了，即使服务器在事务执行完毕之后停机，执行事务所得的结果也不会丢失。

因为Redis的事务不过是简单地用队列包裹起了一组Redis命令，Redis并没有为事务提供任何额外的持久化功能，所以Redis事务的耐久性由Redis所使用的持久化模式决定：

·当服务器在无持久化的内存模式下运作时，事务不具有耐久性：一旦服务器停机，包括事务数据在内的所有服务器数据都将丢失。

·当服务器在RDB持久化模式下运作时，服务器只会在特定的保存条件被满足时，才会执行BGSAVE命令，对数据库进行保存操作，并且异步执行的BGSAVE不能保证事务数据被第一时间保存到硬盘里面，因此RDB持久化模式下的事务也不具有耐久性。

·当服务器运行在AOF持久化模式下，并且appendfsync选项的值为always时，程序总会在执行命令之后调用同步（sync）函数，将命令数据真正地保存到硬盘里面，因此这种配置下的事务是具有耐久性的。

·当服务器运行在AOF持久化模式下，并且appendfsync选项的值为everysec时，程序会每秒同步一次命令数据到硬盘。因为停机可能会恰好发生在等待同步的那一秒钟之内，这可能会造成事务数据丢失，所以这种配置下的事务不具有耐久性。

·当服务器运行在AOF持久化模式下，并且appendfsync选项的值为no时，程序会交由操作系统来决定何时将命令数据同步到硬盘。因为事务数据可能在等待同步的过程中丢失，所以这种配置下的事务不具有耐久性。

no-appendfsync-on-rewrite配置选项对耐久性的影响

配置选项no-appendfsync-on-rewrite可以配合appendfsync选项为always或者everysec的AOF持久化模式使用。当no-appendfsync-on-rewrite选项处于打开状态时，在执行BGSAVE命令或者BGREWRITEAOF命令期间，服务器会暂时停止对AOF文件进行同步，从而尽可能地减少I/O阻塞。但是这样一来，关于“always模式的AOF持久化可以保证事务的耐久性”这一结论将不再成立，因为在服务器停止对AOF文件进行同步期间，事务结果可能会因为停机而丢失。因此，如果服务器打开了no-appendfsync-on-rewrite选项，那么即使服务器运行在always模式的AOF持久化之下，事务也不具有耐久性。在默认配置下，no-appendfsync-on-rewrite处于关闭状态。

不论Redis在什么模式下运作，在一个事务的最后加上SAVE命令总可以保证事务的耐久性：

```
redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> SAVE
QUEUED
redis> EXEC
1) OK
2) OK
```

不过因为这种做法的效率太低，所以并不具有实用性。

19.4 重点回顾

·事务提供了一种将多个命令打包，然后一次性、有序地执行的机制。

·多个命令会被入队到事务队列中，然后按先进先出（FIFO）的顺序执行。

·事务在执行过程中不会被中断，当事务队列中的所有命令都被执行完毕之后，事务才会结束。

·带有WATCH命令的事务会将客户端和被监视的键在数据库的watched_keys字典中进行关联，当键被修改时，程序会将所有监视被修改键的客户端的REDIS_DIRTY_CAS标志打开。

·只有在客户端的REDIS_DIRTY_CAS标志未被打开时，服务器才会执行客户端提交的事务，否则的话，服务器将拒绝执行客户端提交的事务。

·Redis的事务总是具有ACID中的原子性、一致性和隔离性，当服务器运行在AOF持久化模式下，并且appendfsync选项的值为always时，事务也具有耐久性。

19.5 参考资料

·维基百科的ACID词条给出了ACID性质的定义：http://en.wikipedia.org/wiki/ACID。

·《数据库系统实现》一书的第6章《系统故障对策》，对事务、事务错误、日志等主题进行了讨论。

·Redis官方网站上的《事务》文档记录了Redis处理事务错误的方式，以及Redis不支持事务回滚的原因：http://redis.io/topics/transactions。

第20章　Lua脚本

Redis从2.6版本开始引入对Lua脚本的支持，通过在服务器中嵌入Lua环境，Redis客户端可以使用Lua脚本，直接在服务器端原子地执行多个Redis命令。

其中，使用EVAL命令可以直接对输入的脚本进行求值：

```
redis> EVAL "return 'hello world'" 0
"hello world"
```

而使用EVALSHA命令则可以根据脚本的SHA1校验和来对脚本进行求值，但这个命令要求校验和对应的脚本必须至少被EVAL命令执行过一次：

```
redis> EVAL "return 1+1" 0
(integer) 2
redis> EVALSHA "a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9" 0 //
上一个脚本的校验和
integer) 2
```

或者这个校验和对应的脚本曾经被SCRIPT LOAD命令载入过：

```
redis> SCRIPT LOAD "return 2*2"
"4475bfb5919b5ad16424cb50f74d4724ae833e72"
redis> EVALSHA "4475bfb5919b5ad16424cb50f74d4724ae833e72" 0
(integer) 4
```

本章将对Redis服务器中与Lua脚本有关的各个部分进行介绍。

首先，本章将介绍Redis服务器初始化Lua环境的整个过程，说明Redis对Lua环境进行了哪些修改，而这些修改又对用户执行Lua脚本产生了什么影响和限制。

接着，本章将介绍与Lua环境进行协作的两个组件，它们分别是负责执行Lua脚本中包含的Redis命令的伪客户端，以及负责保存传入服务器的Lua脚本的脚本字典。了解伪客户端可以知道脚本中的Redis命令在执行时，服务器与Lua环境的交互过程，而了解脚本字典则有助于理解SCRIPT EXISTS命令和脚本复制功能的实现原理。

在这之后，本章将介绍EVAL命令和EVALSHA命令的实现原理，说明Lua脚本在Redis服务器中是如何被执行的，并对管理脚本的四个命令——SCRIPT FLUSH命令、SCRIPT EXISTS命令、SCRIPT LOAD命令、SCRIPT KILL命令的实现原理进行介绍。

最后，本章将以介绍Redis在主从服务器之间复制Lua脚本的方法作为本章的结束。

20.1　创建并修改Lua环境

为了在Redis服务器中执行Lua脚本，Redis在服务器内嵌了一个Lua环境（environ-ment），并对这个Lua环境进行了一系列修改，从而确保这个Lua环境可以满足Redis服务器的需要。

Redis服务器创建并修改Lua环境的整个过程由以下步骤组成：

1）创建一个基础的Lua环境，之后的所有修改都是针对这个环境进行的。

2）载入多个函数库到Lua环境里面，让Lua脚本可以使用这些函数库来进行数据操作。

3）创建全局表格redis，这个表格包含了对Redis进行操作的函数，比如用于在Lua脚本中执行Redis命令的redis.call函数。

4）使用Redis自制的随机函数来替换Lua原有的带有副作用的随机函数，从而避免在脚本中引入副作用。

5）创建排序辅助函数，Lua环境使用这个辅佐函数来对一部分Redis命令的结果进行排序，从而消除这些命令的不确定性。

6）创建redis.pcall函数的错误报告辅助函数，这个函数可以提供更详细的出错信息。

7）对Lua环境中的全局环境进行保护，防止用户在执行Lua脚本的过程中，将额外的全局变量添加到Lua环境中。

8）将完成修改的Lua环境保存到服务器状态的lua属性中，等待执行服务器传来的Lua脚本。接下来的各个小节将分别介绍这些步骤。

20.1.1　创建Lua环境

在最开始的这一步，服务器首先调用Lua的C　API函数lua_open，创建一个新的Lua环境。

因为lua_open函数创建的只是一个基本的Lua环境，为了让这个Lua环境可以满足Redis的操作要求，接下来服务器将对这个Lua环境进行一系列修改。

20.1.2　载入函数库

Redis修改Lua环境的第一步，就是将以下函数库载入到Lua环境里面：

·基础库（base library）：这个库包含Lua的核心（core）函数，比如assert、error、pairs、tostring、pcall等。另外，为了防止用户从外部文件中引入不安全的代码，库中的loadfile函数会被删除。

·表格库（table library）：这个库包含用于处理表格的通用函数，比如table.concat、table.insert、table.remove、table.sort等。

·字符串库（string library）：这个库包含用于处理字符串的通用函数，比如用于对字符串进行查找的string.find函数，对字符串进行格式化的string.format函数，查看字符串长度的string.len函数，对字符串进行翻转的string.reverse函数等。

·数学库（math library）：这个库是标准C语言数学库的接口，它包括计算绝对值的math.abs函数，返回多个数中的最大值和最小值的math.max函数和math.min函数，计算二次方根的math.sqrt函数，计算对数的math.log函数等。

·调试库（debug library）：这个库提供了对程序进行调试所需的函数，比如对程序设置钩子和取得钩子的debug.sethook函数和debug.gethook函数，返回给定函数相关信息的debug.getinfo函数，为对象设置元数据的debug.setmetatable函数，获取对象元数据的debug.getmetatable函数等。

·Lua CJSON库（http://www.kyne.com.au/~mark/software/lua-cjson.php）：这个库用于处理UTF-8编码的JSON格式，其中cjson.decode函数将一个JSON格式的字符串转换为一个Lua值，而cjson.encode函数将一个Lua值序列化为JSON格式的字符串。

·Struct库（http://www.inf.puc-rio.br/~roberto/struct/）：这个库用于在Lua值和C结构（struct）之间进行转换，函数struct.pack将多个Lua值打包成一个类结构（struct-like）字符串，而函数struct.unpack则从一个类结构字符串中解包出多个Lua值。

·Lua cmsgpack库（https://github.com/antirez/lua-cmsgpack）：这个库用于处理MessagePack格式的数据，其中cmsgpack.pack函数将Lua值转换为MessagePack数据，而cmsgpack.unpack函数则将MessagePack数据转换为Lua值。

通过使用这些功能强大的函数库，Lua脚本可以直接对执行Redis命令获得的数据进行复杂的操作。

20.1.3 创建redis全局表格

在这一步，服务器将在Lua环境中创建一个redis表格（table），并将它设为全局变量。这个redis表格包含以下函数：

·用于执行Redis命令的redis.call和redis.pcall函数。

·用于记录Redis日志（log）的redis.log函数，以及相应的日志级别（level）常量：redis.LOG_DEBUG，redis.LOG_VERBOSE，redis.LOG_NOTICE，以及redis.LOG_WARNING。

·用于计算SHA1校验和的redis.sha1hex函数。

·用于返回错误信息的redis.error_reply函数和redis.status_reply函数。

在这些函数里面，最常用也最重要的要数redis.call函数和redis.pcall函数，通过这两个函数，用户可以直接在Lua脚本中执行Redis命令：

```
redis> EVAL "return redis.call('PING')" 0
PONG
```

20.1.4　使用Redis自制的随机函数来替换Lua原有的随机函数

为了保证相同的脚本可以在不同的机器上产生相同的结果，Redis要求所有传入服务器的Lua脚本，以及Lua环境中的所有函数，都必须是无副作用（side effect）的纯函数（pure function）。

但是，在之前载入Lua环境的math函数库中，用于生成随机数的math.random函数和math.randomseed函数都是带有副作用的，它们不符合Redis对Lua环境的无副作用要求。

因为这个原因，Redis使用自制的函数替换了math库中原有的math.random函数和math.randomseed函数，替换之后的两个函数有以下特征：

·对于相同的seed来说，math.random总产生相同的随机数序列，这个函数是一个纯函数。

·除非在脚本中使用math.randomseed显式地修改seed，否则每次运行脚本时，Lua环境都使用固定的math.randomseed（0）语句来初始化seed。

例如，使用以下脚本，我们可以打印seed值为0时，math.random对于输入10至1所产生的随机序列：

```
--random-with-default-seed.lua
local i = 10
local seq = {}
while (i > 0) do
```

```
    seq[i] = math.random(i)
    i = i-1
end
return seq
```

无论执行这个脚本多少次，产生的值都是相同的：

```
$ redis-cli --eval random-with-default-seed.lua
1) (integer) 1
2) (integer) 2
3) (integer) 2
4) (integer) 3
5) (integer) 4
6) (integer) 4
7) (integer) 7
8) (integer) 1
9) (integer) 7
10) (integer) 2
```

但是，如果我们在另一个脚本里面，调用math.randomseed将seed修改为10086：

```
--random-with-new-seed.lua
math.randomseed(10086) --change seed
local i = 10
local seq = {}
while (i > 0) do
    seq[i] = math.random(i)
    i = i-1
end
return seq
```

那么这个脚本生成的随机数序列将和使用默认seed值0时生成的随机序列不同：

```
$ redis-cli --eval random-with-new-seed.lua
1) (integer) 1
2) (integer) 1
3) (integer) 2
4) (integer) 1
5) (integer) 1
6) (integer) 3
7) (integer) 1
8) (integer) 1
9) (integer) 3
10) (integer) 1
```

20.1.5 创建排序辅助函数

上一个小节说到，为了防止带有副作用的函数令脚本产生不一致的数据，Redis对math库的math.random函数和math.randomseed函数进行了替换。

对于Lua脚本来说，另一个可能产生不一致数据的地方是那些带有不确定性质的命令。比如对于一个集合键来说，因为集合元素的排列是无序的，所以即使两个集合的元素完全相同，它们的输出结果也可能并不相同。

考虑下面这个集合例子：

```
redis> SADD fruit apple banana cherry
(integer) 3
redis> SMEMBERS fruit
1) "cherry"
2) "banana"
3) "apple"
redis> SADD another-fruit cherry banana apple
(integer) 3
redis> SMEMBERS another-fruit
1) "apple"
2) "banana"
3) "cherry"
```

这个例子中的fruit集合和another-fruit集合包含的元素是完全相同的，只是因为集合添加元素的顺序不同，SMEMBERS命令的输出就产生了不同的结果。

Redis将SMEMBERS这种在相同数据集上可能会产生不同输出的命令称为“带有不确定性的命令”，这些命令包括：

·SINTER

·SUNION

·SDIFF

·SMEMBERS

·HKEYS

·HVALS

·KEYS

为了消除这些命令带来的不确定性，服务器会为Lua环境创建一个排序辅助函数_redis_compare_helper，当Lua脚本执行完一个带有不确定性的命令之后，程序会使用_redis_compare_helper作为对比函数，自动调用table.sort函数对命令的返回值做一次排序，以此来保证相同的数据集总是产生相同的输出。

举个例子，如果我们在Lua脚本中对fruit集合和another-fruit集合执行SMEMBERS命令，那么两个脚本将得出相同的结果，因为脚本已经对SMEMBERS命令的输出进行过排序了：

```
redis> EVAL "return redis.call('SMEMBERS', KEYS[1])" 1 fruit
1) "apple"
2) "banana"
3) "cherry"
redis> EVAL "return redis.call('SMEMBERS', KEYS[1])" 1 another-fruit
1) "apple"
```

```
2) "banana"
3) "cherry"
```

20.1.6　创建redis.pcall函数的错误报告辅助函数

在这一步，服务器将为Lua环境创建一个名为_redis_err_handler的错误处理函数，当脚本调用redis.pcall函数执行Redis命令，并且被执行的命令出现错误时，_redis_err_handler就会打印出错代码的来源和发生错误的行数，为程序的调试提供方便。

举个例子，如果客户端要求服务器执行以下Lua脚本：

```
--
第1
行
--
第2
行
--
第3
行
return redis.pcall('wrong command')
```

那么服务器将向客户端返回一个错误：

```
$ redis-cli --eval wrong-command.lua
(error) @user_script: 4: Unknown Redis command called from Lua script
```

其中@user_script说明这是一个用户定义的函数，而之后的4则说明出错的代码位于Lua脚本的第四行。

20.1.7　保护Lua的全局环境

在这一步，服务器将对Lua环境中的全局环境进行保护，确保传入服务器的脚本不会因为忘记使用local关键字而将额外的全局变量添加到Lua环境里面。

因为全局变量保护的原因，当一个脚本试图创建一个全局变量时，服务器将报告一个错误：

```
redis> EVAL "x = 10" 0
(error) ERR Error running script
(call to f_df1ad3745c2d2f078f0f41377a92bb6f8ac79af0):
@enable_strict_lua:7: user_script:1:
Script attempted to create global variable 'x'
```

除此之外，试图获取一个不存在的全局变量也会引发一个错误：

```
redis> EVAL "return x" 0
(error) ERR Error running script
(call to f_03c387736bb5cc009ff35151572cee04677aa374):
@enable_strict_lua:14: user_script:1:
```

```
Script attempted to access unexisting global variable 'x'
```

不过Redis并未禁止用户修改已存在的全局变量，所以在执行Lua脚本的时候，必须非常小心，以免错误地修改了已存在的全局变量：

```
redis> EVAL "redis = 10086; return redis" 0
(integer) 10086
```

20.1.8　将Lua环境保存到服务器状态的lua属性里面

经过以上的一系列修改，Redis服务器对Lua环境的修改工作到此就结束了，在最后的这一步，服务器会将Lua环境和服务器状态的lua属性关联起来，如图20-1所示。

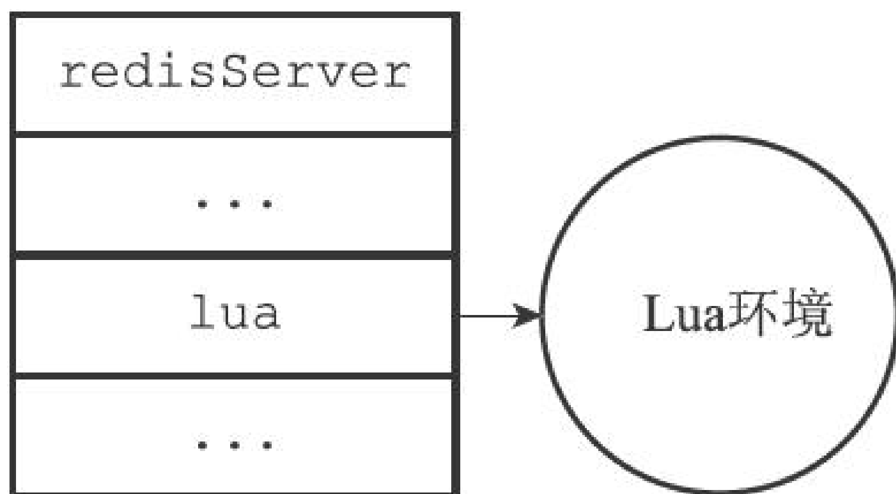


图20-1　服务器状态中的Lua环境

因为Redis使用串行化的方式来执行Redis命令，所以在任何特定时间里，最多都只会有一个脚本能够被放进Lua环境里面运行，因此，整个Redis服务器只需要创建一个Lua环境即可。

20.2 Lua环境协作组件

除了创建并修改Lua环境之外，Redis服务器还创建了两个用于与Lua环境进行协作的组件，它们分别是负责执行Lua脚本中的Redis命令的伪客户端，以及用于保存Lua脚本的lua_scripts字典。

接下来的两个小节将分别介绍这两个组件。

20.2.1 伪客户端

因为执行Redis命令必须有相应的客户端状态，所以为了执行Lua脚本中包含的Redis命令，Redis服务器专门为Lua环境创建了一个伪客户端，并由这个伪客户端负责处理Lua脚本中包含的所有Redis命令。

Lua脚本使用redis.call函数或者redis.pcall函数执行一个Redis命令，需要完成以下步骤：

1）Lua环境将redis.call函数或者redis.pcall函数想要执行的命令传给伪客户端。

2）伪客户端将脚本想要执行的命令传给命令执行器。

3）命令执行器执行伪客户端传给它的命令，并将命令的执行结果返回给伪客户端。

4）伪客户端接收命令执行器返回的命令结果，并将这个命令结果返回给Lua环境。

5）Lua环境在接收到命令结果之后，将该结果返回给redis.call函数或者redis.pcall函数。

6）接收到结果的redis.call函数或者redis.pcall函数会将命令结果作为函数返回值返回给脚本中的调用者。

图20-2展示了Lua脚本在调用redis.call函数时，Lua环境、伪客户端、命令执行器三者之间的通信过程（调用redis.pcall函数时产生的通信过程也是一样的）。

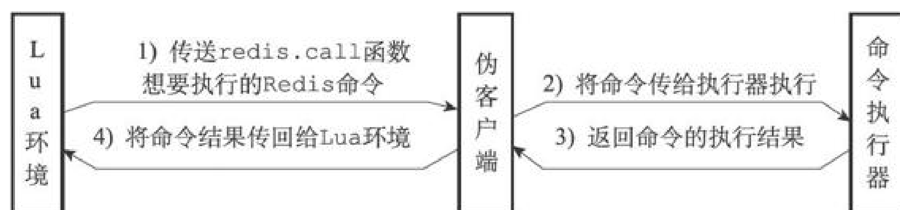


图20-2　Lua脚本执行Redis命令时的通信步骤

举个例子，图20-3展示了Lua脚本在执行以下命令时：

```
redis> EVAL "return redis.call('DBSIZE')" 0
(integer) 10086
```

Lua环境、伪客户端、命令执行器三者之间的通信过程。

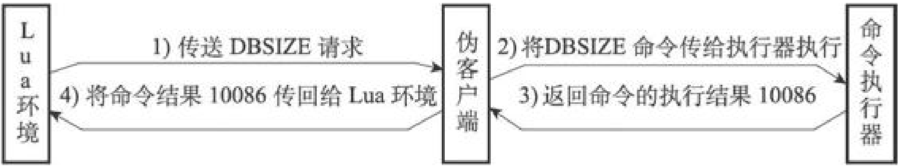


图20-3　Lua脚本执行DBSIZE命令时的通信步骤

20.2.2　lua_scripts字典

除了伪客户端之外，Redis服务器为Lua环境创建的另一个协作组件是lua_scripts字典，这个字典的键为某个Lua脚本的SHA1校验和（checksum），而字典的值则是SHA1校验和对应的Lua脚本：

```
struct redisServer {
    // ...
    dict *lua_scripts;
    // ...
};
```

Redis服务器会将所有被EVAL命令执行过的Lua脚本，以及所有被SCRIPT LOAD命令载入过的Lua脚本都保存到lua_scripts字典里面。

举个例子，如果客户端向服务器发送以下命令：

```
redis> SCRIPT LOAD "return 'hi'"
"2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
redis> SCRIPT LOAD "return 1+1"
"a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9"
redis> SCRIPT LOAD "return 2*2"
"4475bfb5919b5ad16424cb50f74d4724ae833e72"
```

那么服务器的lua_scripts字典将包含被SCRIPT　LOAD命令载入的三个Lua脚本，如图20-4所示。

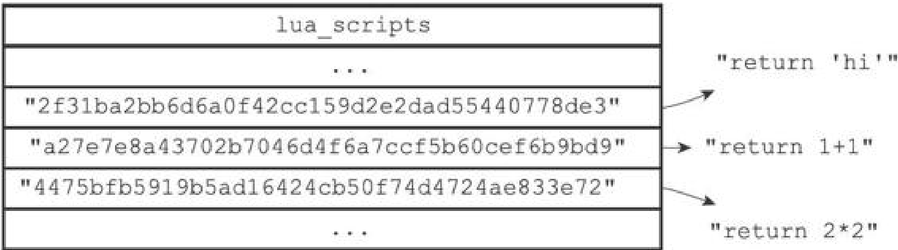


图20-4　lua_scripts字典示例

lua_scripts字典有两个作用，一个是实现SCRIPT EXISTS命令，另一个是实现脚本复制功能，本章稍后将详细说明lua_scripts字典在这两个功能中的作用。

20.3　EVAL命令的实现

EVAL命令的执行过程可以分为以下三个步骤：

1）根据客户端给定的Lua脚本，在Lua环境中定义一个Lua函数。

2）将客户端给定的脚本保存到lua_scripts字典，等待将来进一步使用。

3）执行刚刚在Lua环境中定义的函数，以此来执行客户端给定的Lua脚本。

以下三个小节将以：

```
redis> EVAL "return 'hello world'" 0
"hello world"
```

命令为示例，分别介绍EVAL命令执行的三个步骤。

20.3.1　定义脚本函数

当客户端向服务器发送EVAL命令，要求执行某个Lua脚本的时候，服务器首先要做的就是在Lua环境中，为传入的脚本定义一个与这个脚本相对应的Lua函数，其中，Lua函数的名字由f_前缀加上脚本的SHA1校验和（四十个字符长）组成，而函数的体（body）则是脚本本身。

举个例子，对于命令：

```
EVAL "return 'hello world'" 0
```

来说，服务器将在Lua环境中定义以下函数：

```
function f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91()
    return 'hello world'
end
```

因为客户端传入的脚本为return'hello　world'，而这个脚本的SHA1校验和为5332031c6b470dc5a0dd9b4bf2030dea6d65de91，所以函数的名字为f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91，而函数的体则为return'hello world'。

使用函数来保存客户端传入的脚本有以下好处：

·执行脚本的步骤非常简单，只要调用与脚本相对应的函数即可。

·通过函数的局部性来让Lua环境保持清洁，减少了垃圾回收的工作

量，并且避免了使用全局变量。

·如果某个脚本所对应的函数在Lua环境中被定义过至少一次，那么只要记得这个脚本的SHA1校验和，服务器就可以在不知道脚本本身的情况下，直接通过调用Lua函数来执行脚本，这是EVALSHA命令的实现原理，稍后在介绍EVALSHA命令的实现时就会说到这一点。

20.3.2　将脚本保存到lua_scripts字典

EVAL命令要做的第二件事是将客户端传入的脚本保存到服务器的lua_scripts字典里面。举个例子，对于命令：

```
EVAL "return 'hello world'" 0
```

来说，服务器将在lua_scripts字典中新添加一个键值对，其中键为Lua脚本的SHA1校验和：

```
5332031c6b470dc5a0dd9b4bf2030dea6d65de91
```

而值则为Lua脚本本身：

```
return 'hello world'
```

添加新键值对之后的lua_scripts字典如图20-5所示。

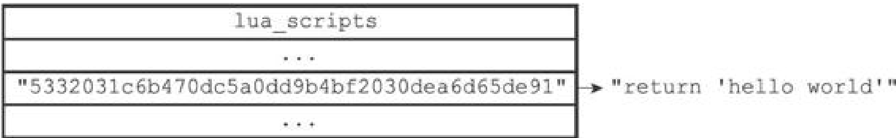


图20-5　添加新键值对之后的lua_scripts字典

20.3.3　执行脚本函数

在为脚本定义函数，并且将脚本保存到lua_scripts字典之后，服务器还需要进行一些设置钩子、传入参数之类的准备动作，才能正式开始执行脚本。

整个准备和执行脚本的过程如下：

1）将EVAL命令中传入的键名（key name）参数和脚本参数分别保存到KEYS数组和ARGV数组，然后将这两个数组作为全局变量传入到Lua环境里面。

2）为Lua环境装载超时处理钩子（hook），这个钩子可以在脚本出现超时运行情况时，让客户端通过SCRIPT　KILL命令停止脚本，或者通

过SHUTDOWN命令直接关闭服务器。

3）执行脚本函数。

4）移除之前装载的超时钩子。

5）将执行脚本函数所得的结果保存到客户端状态的输出缓冲区里面，等待服务器将结果返回给客户端。

6）对Lua环境执行垃圾回收操作。

举个例子，对于如下命令：

```
EVAL "return 'hello world'" 0
```

服务器将执行以下动作：

1）因为这个脚本没有给定任何键名参数或者脚本参数，所以服务器会跳过传值到KEYS数组或ARGV数组这一步。

2）为Lua环境装载超时处理钩子。

3）在Lua环境中执行f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91函数。

4）移除超时钩子。

5）将执行f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91函数所得的结果"hello world"保存到客户端状态的输出缓冲区里面。

6）对Lua环境执行垃圾回收操作。

至此，命令：

```
EVAL "return 'hello world'" 0
```

执行算是告一段落，之后服务器只要将保存在输出缓冲区里面的执行结果返回给执行EVAL命令的客户端就可以了。

20.4　EVALSHA命令的实现

本章前面介绍EVAL命令的实现时说过，每个被EVAL命令成功执行过的Lua脚本，在Lua环境里面都有一个与这个脚本相对应的Lua函数，函数的名字由f_前缀加上40个字符长的SHA1校验和组成，例如f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91。

只要脚本对应的函数曾经在Lua环境里面定义过，那么即使不知道脚本的内容本身，客户端也可以根据脚本的SHA1校验和来调用脚本对应的函数，从而达到执行脚本的目的，这就是EVALSHA命令的实现原理。

可以用伪代码来描述这一原理：

```
def EVALSHA(sha1):
    #
拼接出函数的名字
    #
例如：f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91
    func_name = "f_" + sha1
    #
查看这个函数在Lua
环境中是否存在
    if function_exists_in_lua_env(func_name):
        #
如果函数存在，那么执行它
        execute_lua_function(func_name)
    else:
        #
如果函数不存在，那么返回一个错误
        send_script_error("SCRIPT NOT FOUND")
```

举个例子，当服务器执行完以下EVAL命令之后：

```
redis> EVAL "return 'hello world'" 0
"hello world"
```

Lua环境里面就定义了以下函数：

```
function f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91()
    return 'hello world'
end
```

当客户端执行以下EVALSHA命令时：

```
redis> EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
"hello world"
```

服务器首先根据客户端输入的SHA1校验和，检查函数f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91是否存在于Lua环境中，得到的回应是该函数确实存在，于是服务器执行Lua环境中的

f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91函数，并将结果"hello world"返回给客户端。

20.5 脚本管理命令的实现

除了EVAL命令和EVALSHA命令之外，Redis中与Lua脚本有关的命令还有四个，它们分别是SCRIPT FLUSH命令、SCRIPT EXISTS命令、SCRIPT LOAD命令、以及SCRIPT KILL命令。

接下来的四个小节将分别对这四个命令的实现原理进行介绍。

20.5.1 SCRIPT FLUSH

SCRIPT FLUSH命令用于清除服务器中所有和Lua脚本有关的信息，这个命令会释放并重建lua_scripts字典，关闭现有的Lua环境并重新创建一个新的Lua环境。

以下为SCRIPT FLUSH命令的实现伪代码：

```
def SCRIPT_FLUSH():
    #
释放脚本字典
    dictRelease(server.lua_scripts)
    #
重建脚本字典
    server.lua_scripts = dictCreate(...)
    #
关闭Lua
环境
    lua_close(server.lua)
    #
初始化一个新的Lua
环境
    server.lua = init_lua_env()
```

20.5.2 SCRIPT EXISTS

SCRIPT EXISTS命令根据输入的SHA1校验和，检查校验和对应的脚本是否存在于服务器中。

SCRIPT EXISTS命令是通过检查给定的校验和是否存在于lua_scripts字典来实现的，以下是该命令的实现伪代码：

```
def SCRIPT_EXISTS(*sha1_list):
    #
结果列表
    result_list = []
    #
遍历输入的所有SHA1
校验和
    for sha1 in sha1_list:
        #
检查校验和是否为lua_scripts
字典的键
        #
如果是的话，那么表示校验和对应的脚本存在
        #
```

```
否则的话，脚本就不存在
        if sha1 in server.lua_scripts:
            #
存在用1
表示
            result_list.append(1)
        else:
            #
不存在用0
表示
            result_list.append(0)
    #
向客户端返回结果列表
    send_list_reply(result_list)
```

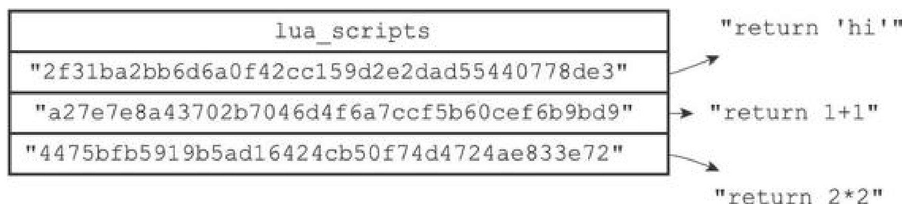
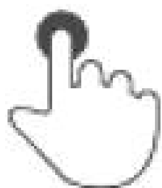


图20-6　lua_scripts字典

举个例子，对于图20-6所示的lua_scripts字典来说，我们可以进行以下测试：

```
redis> SCRIPT EXISTS "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
1) (integer) 1
redis> SCRIPT EXISTS "a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9"
1) (integer) 1
redis> SCRIPT EXISTS "4475bfb5919b5ad16424cb50f74d4724ae833e72"
1) (integer) 1
redis> SCRIPT EXISTS "NotExistsScriptSha1HereABCDEFGHIJKLMNOPQ"
1) (integer) 0
```

从测试结果可知，除了最后一个校验和之外，其他校验和对应的脚本都存在于服务器中。



注意

SCRIPT EXISTS命令允许一次传入多个SHA1校验和，不过因为SHA1校验和太长，所以示例里分开多次来进行测试。

实现SCRIPT EXISTS实际上并不需要lua_scripts字典的值。如果lua_scripts字典只用于实现SCRIPT EXISTS命令的话，那么字典只需要保存Lua脚本的SHA1校验和就可以了，并不需要保存Lua脚本本身。lua_scripts字典既保存脚本的SHA1校验和，又保存脚本本身的原因是为了实现脚本复制功能，详细的情况请看本章稍后对脚本复制功能实现原理的介绍。

20.5.3 SCRIPT LOAD

SCRIPT LOAD命令所做的事情和EVAL命令执行脚本时所做的前两步完全一样：命令首先在Lua环境中为脚本创建相对应的函数，然后再将脚本保存到lua_scripts字典里面。

举个例子，如果我们执行以下命令：

```
redis> SCRIPT LOAD "return 'hi'"
"2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
```

那么服务器将在Lua环境中创建以下函数：

```
function f_2f31ba2bb6d6a0f42cc159d2e2dad55440778de3()
    return 'hi'
end
```

并将键为"2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"，值为"return'hi'"的键值对添加到服务器的lua_scripts字典里面，如图20-7所示。

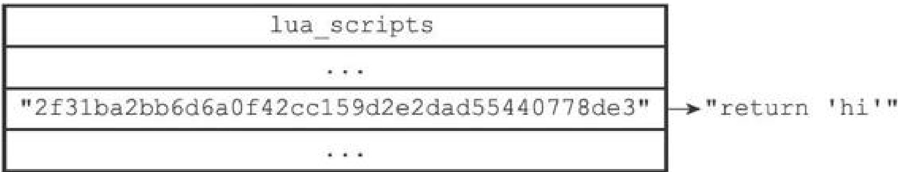


图20-7 lua_scripts字典

完成了这些步骤之后，客户端就可以使用EVALSHA命令来执行前面被SCRIPT LOAD命令载入的脚本了：

```
redis> EVALSHA "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3" 0
"hi"
```

20.5.4 SCRIPT KILL

如果服务器设置了lua-time-limit配置选项，那么在每次执行Lua脚本之前，服务器都会在Lua环境里面设置一个超时处理钩子（hook）。

超时处理钩子在脚本运行期间，会定期检查脚本已经运行了多长时间，一旦钩子发现脚本的运行时间已经超过了lua-time-limit选项设置的时长，钩子将定期在脚本运行的间隙中，查看是否有SCRIPT KILL命令或者SHUTDOWN命令到达服务器。

图20-8展示了带有超时处理钩子的脚本的运行过程。

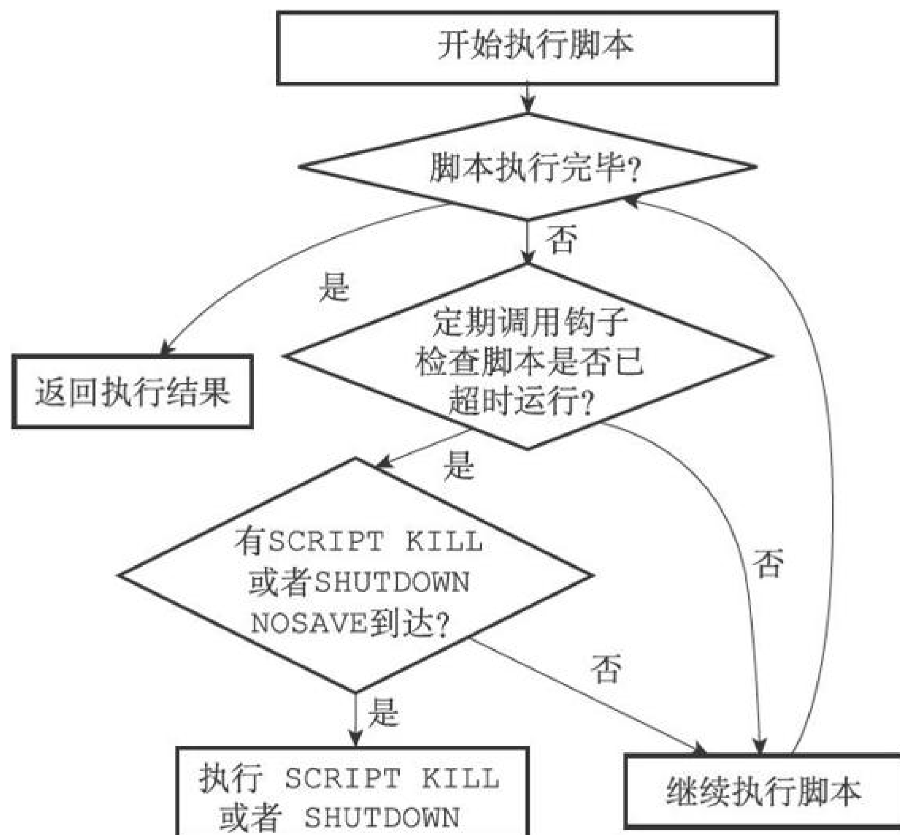


图20-8　带有超时处理钩子的脚本的执行过程

如果超时运行的脚本未执行过任何写入操作，那么客户端可以通过SCRIPT　KILL命令来指示服务器停止执行这个脚本，并向执行该脚本的客户端发送一个错误回复。处理完SCRIPT　KILL命令之后，服务器可以继续运行。

另一方面，如果脚本已经执行过写入操作，那么客户端只能用SHUTDOWN　nosave命令来停止服务器，从而防止不合法的数据被写入数据库中。

20.6 脚本复制

与其他普通Redis命令一样，当服务器运行在复制模式之下时，具有写性质的脚本命令也会被复制到从服务器，这些命令包括EVAL命令、EVALSHA命令、SCRIPT FLUSH命令，以及SCRIPT LOAD命令。

接下来的两个小节将分别介绍这四个命令的复制方法。

20.6.1 复制EVAL命令、SCRIPT FLUSH命令和SCRIPT LOAD命令

Redis复制EVAL、SCRIPT FLUSH、SCRIPT LOAD三个命令的方法和复制其他普通Redis命令的方法一样，当主服务器执行完以上三个命令的其中一个时，主服务器会直接将被执行的命令传播（propagate）给所有从服务器，如图20-9所示。

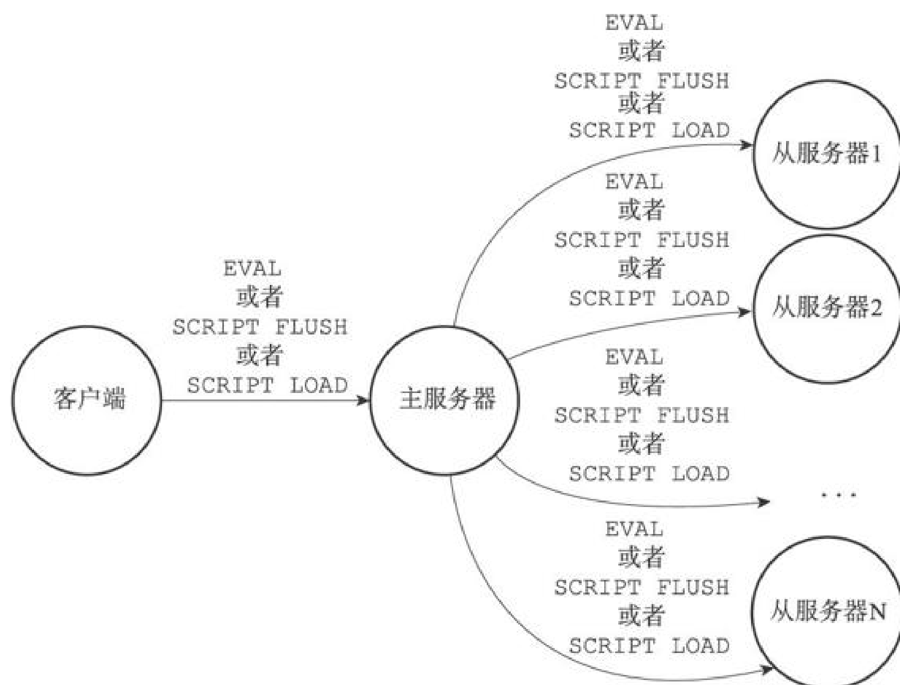


图20-9 将脚本命令传播给从服务器

1.EVAL

对于EVAL命令来说，在主服务器执行的Lua脚本同样会在所有从服务器中执行。

举个例子，如果客户端向主服务器执行以下命令：

```
redis> EVAL "return redis.call('SET', KEYS[1], ARGV[1])" 1 "msg" "hello world"
OK
```

那么主服务器在执行这个EVAL命令之后，将向所有从服务器传播这条EVAL命令，从服务器会接收并执行这条EVAL命令，最终结果是，主从服务器双方都会将数据库"msg"键的值设置为"hello world"，并且将脚本：

```
"return redis.call('SET', KEYS[1], ARGV[1])"
```

保存在脚本字典里面。

2.SCRIPT FLUSH

如果客户端向主服务器发送SCRIPT FLUSH命令，那么主服务器也会向所有从服务器传播SCRIPT FLUSH命令。

最终的结果是，主从服务器双方都会重置自己的Lua环境，并清空自己的脚本字典。

3.SCRIPT LOAD

如果客户端使用SCRIPT LOAD命令，向主服务器载入一个Lua脚本，那么主服务器将向所有从服务器传播相同的SCRIPT LOAD命令，使得所有从服务器也会载入相同的Lua脚本。

举个例子，如果客户端向主服务器发送命令：

```
redis> SCRIPT LOAD "return 'hello world'"
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"
```

那么主服务器也会向所有从服务器传播同样的命令：

```
SCRIPT LOAD "return 'hello world'"
```

最终的结果是，主从服务器双方都会载入脚本：

```
"return 'hello world'"
```

20.6.2 复制EVALSHA命令

EVALSHA命令是所有与Lua脚本有关的命令中，复制操作最复杂的一个，因为主服务器与从服务器载入Lua脚本的情况可能有所不同，所以主服务器不能像复制EVAL命令、SCRIPT LOAD命令或者SCRIPT FLUSH命令那样，直接将EVALSHA命令传播给从服务器。对于一个在主服务器被成功执行的EVALSHA命令来说，相同的EVALSHA命令在从服

务器执行时却可能会出现脚本未找到（not found）错误。

举个例子，假设现在有一个主服务器master，如果客户端向主服务器发送命令：

```
master> SCRIPT LOAD "return 'hello world'"
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"
```

那么在执行这个SCRIPT LOAD命令之后，SHA1值为5332031c6b470dc5a0dd9b4bf2030dea6d65de91的脚本就存在于主服务器中了。

现在，假设一个从服务器slave1开始复制主服务器master，如果master不想办法将脚本：

```
"return 'hello world'"
```

传送给slave1载入的话，那么当客户端向主服务器发送命令：

```
master> EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
"hello world"
```

的时候，master将成功执行这个EVALSHA命令，而当master将这个命令传播给slave1执行的时候，slave1却会出现脚本未找到错误：

```
slave1> EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
(error) NOSCRIPT No matching script. Please use EVAL.
```

更为复杂的是，因为多个从服务器之间载入Lua脚本的情况也可能各有不同，所以即使一个EVALSHA命令可以在某个从服务器成功执行，也不代表这个EVALSHA命令就一定可以在另一个从服务器成功执行。

举个例子，假设有主服务器master和从服务器slave1，并且slave1一直复制着master，所以master载入的所有Lua脚本，slave1也有载入（通过传播EVAL命令或者SCRIPT LOAD命令来实现）。

例如说，如果客户端向master发送命令：

```
master> SCRIPT LOAD "return 'hello world'"
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"
```

那么这个命令也会被传播到slave1上面，所以master和slave1都会成功载入SHA1校验和为5332031c6b470dc5a0dd9b4bf2030dea6d65de91的Lua脚本。

如果这时，一个新的从服务器slave2开始复制主服务器master，如果master不想办法将脚本：

```
"return 'hello world'"
```

传送给slave2的话，那么当客户端向主服务器发送命令：

```
master> EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
"hello world"
```

的时候，master和slave1都将成功执行这个EVALSHA命令，而slave2却会发生脚本未找到错误。

为了防止以上假设的情况出现，Redis要求主服务器在传播EVALSHA命令的时候，必须确保EVALSHA命令要执行的脚本已经被所有从服务器载入过，如果不能确保这一点的话，主服务器会将EVALSHA命令转换成一个等价的EVAL命令，然后通过传播EVAL命令来代替EVALSHA命令。

传播EVALSHA命令，或者将EVALSHA命令转换成EVAL命令，都需要用到服务器状态的lua_scripts字典和repl_scriptcache_dict字典，接下来的小节将分别介绍这两个字典的作用，并最终说明Redis复制EVALSHA命令的方法。

1.判断传播EVALSHA命令是否安全的方法

主服务器使用服务器状态的repl_scriptcache_dict字典记录自己已经将哪些脚本传播给了所有从服务器：

```
struct redisServer {
    // ...
    dict *repl_scriptcache_dict;
    // ...
};
```

repl_scriptcache_dict字典的键是一个个Lua脚本的SHA1校验和，而字典的值则全部都是NULL，当一个校验和出现在repl_scriptcache_dict字典时，说明这个校验和对应的Lua脚本已经传播给了所有从服务器，主服务器可以直接向从服务器传播包含这个SHA1校验和的EVALSHA命令，而不必担心从服务器会出现脚本未找到错误。

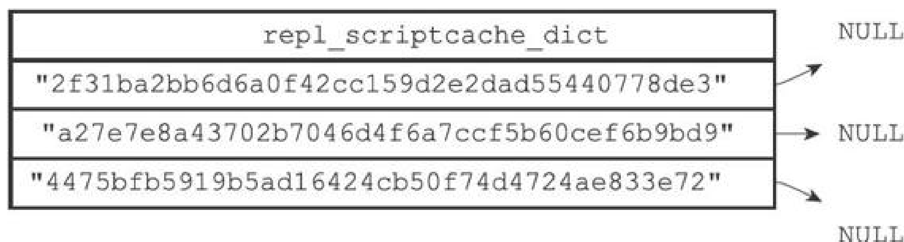


图20-10　一个repl_scriptcache_dict字典示例

举个例子，如果主服务器repl_scriptcache_dict字典的当前状态如图20-10所示，那么主服务器可以向从服务器传播以下三个EVALSHA命令，并且从服务器在执行这些EVALSHA命令的时候不会出现脚本未找到错误：

```
EVALSHA "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3" ...
EVALSHA "a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9" ...
EVALSHA "4475bfb5919b5ad16424cb50f74d4724ae833e72" ...
```

另一方面，如果一个脚本的SHA1校验和存在于lua_scripts字典，但是却不存在于repl_scriptcache_dict字典，那么说明校验和对应的Lua脚本已经被主服务器载入，但是并没有传播给所有从服务器，如果我们尝试向从服务器传播包含这个SHA1校验和的EVALSHA命令，那么至少有一个从服务器会出现脚本未找到错误。

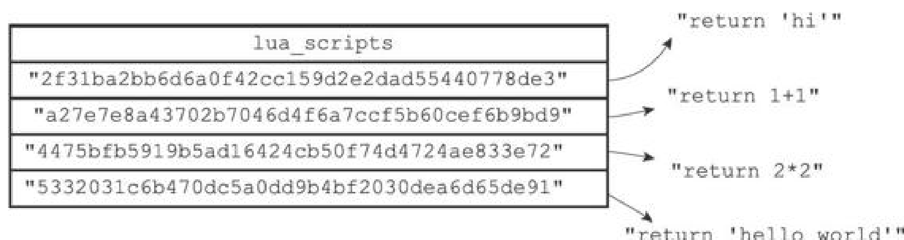


图20-11　lua_scripts字典

举个例子，对于图20-11所示的lua_scripts字典，以及图20-10所示的repl_scriptcache_dict字典来说，SHA1校验和为：

```
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"
```

的脚本：

```
"return 'hello world'"
```

虽然存在于lua_scripts字典，但是repl_scriptcache_dict字典却并不包含校验和"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"，这说明脚本：

```
"return 'hello world'"
```

虽然已经载入到主服务器里面，但并未传播给所有从服务器，如果主服务器尝试向从服务器发送命令：

```
EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" ...
```

那么至少会有一个从服务器遇上脚本未找到错误。

2.清空repl_scriptcache_dict字典

每当主服务器添加一个新的从服务器时，主服务器都会清空自己的repl_scriptcache_dict字典，这是因为随着新从服务器的出现，repl_scriptcache_dict字典里面记录的脚本已经不再被所有从服务器载入过，所以主服务器会清空repl_scriptcache_dict字典，强制自己重新向所有从服务器传播脚本，从而确保新的从服务器不会出现脚本未找到错误。

3.EVALSHA命令转换成EVAL命令的方法

通过使用EVALSHA命令指定的SHA1校验和，以及lua_scripts字典保存的Lua脚本，服务器总可以将一个EVALSHA命令：

```
EVALSHA <sha1> <numkeys> [key ...] [arg ...]
```

转换成一个等价的EVAL命令：

```
EVAL <script> <numkeys> [key ...] [arg ...]
```

具体的转换方法如下：

1）根据SHA1校验和sha1，在lua_scripts字典中查找sha1对应的Lua脚本script。

2）将原来的EVALSHA命令请求改写成EVAL命令请求，并且将校验和sha1改成脚本script，至于numkeys、key、arg等参数则保持不变。

举个例子，对于图20-11所示的lua_scripts字典，以及图20-10所示的repl_scriptcache_dict字典来说，我们总可以将命令：

```
EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
```

改写成命令：

```
EVAL "return 'hello world'" 0
```

其中脚本的内容：

```
"return 'hello world'"
```

来源于lua_scripts字典“5332031c6b470dc5a0dd9b4bf2030dea6d65de91”键的值。

如果一个SHA1值所对应的Lua脚本没有被所有从服务器载入过，那么主服务器可以将EVALSHA命令转换成等价的EVAL命令，然后通过传

播等价的EVAL命令来代替原本想要传播的EVALSHA命令，以此来产生相同的脚本执行效果，并确保所有从服务器都不会出现脚本未找到错误。

另外，因为主服务器在传播完EVAL命令之后，会将被传播脚本的SHA1校验和（也即是原本EVALSHA命令指定的那个校验和）添加到repl_scriptcache_dict字典里面，如果之后EVALSHA命令再次指定这个SHA1校验和，主服务器就可以直接传播EVALSHA命令，而不必再次对EVALSHA命令进行转换。

4.传播EVALSHA命令的方法

当主服务器成功在本机执行完一个EVALSHA命令之后，它将根据EVALSHA命令指定的SHA1校验和是否存在于repl_scriptcache_dict字典来决定是向从服务器传播EVALSHA命令还是EVAL命令：

1）如果EVALSHA命令指定的SHA1校验和存在于repl_scriptcache_dict字典，那么主服务器直接向从服务器传播EVALSHA命令。

2）如果EVALSHA命令指定的SHA1校验和不存在于repl_scriptcache_dict字典，那么主服务器会将EVALSHA命令转换成等价的EVAL命令，然后传播这个等价的EVAL命令，并将EVALSHA命令指定的SHA1校验和添加到repl_scriptcache_dict字典里面。

图20-12展示了这个判断过程。

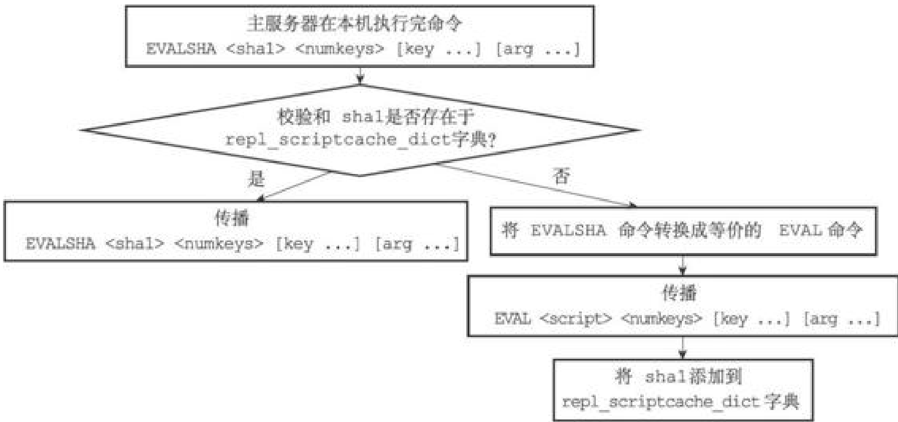


图20-12　主服务器判断传播EVAL还是EVALSHA的过程

举个例子，假设服务器当前lua_scripts字典和repl_scriptcache_dict字典的状态如图20-13所示，如果客户端向主服务器发送命令：

```
EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
```

那么主服务器在执行完这个EVALSHA命令之后，会将这个EVALSHA命令转换成等价的EVAL命令：

```
EVAL "return 'hello world'" 0
```

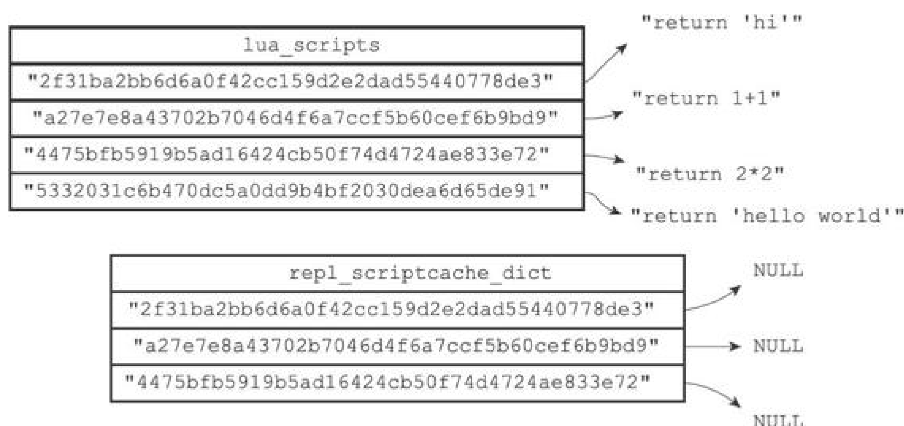


图20-13　执行EVALSHA命令之前的lua_scripts字典和repl_scriptcache_dict字典

并向所有从服务器传播这个EVAL命令。

除此之外，主服务器还会将SHA1校验和"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"添加到repl_scriptcache_dict字典里，这样当客户端下次再发送命令：

```
EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
```

的时候，主服务器就可以直接向从服务器传播这个EVALSHA命令，而无须将EVALSHA命令转换成EVAL命令再传播。

添加"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"之后的repl_scriptcac-he_dict字典如图20-14所示。

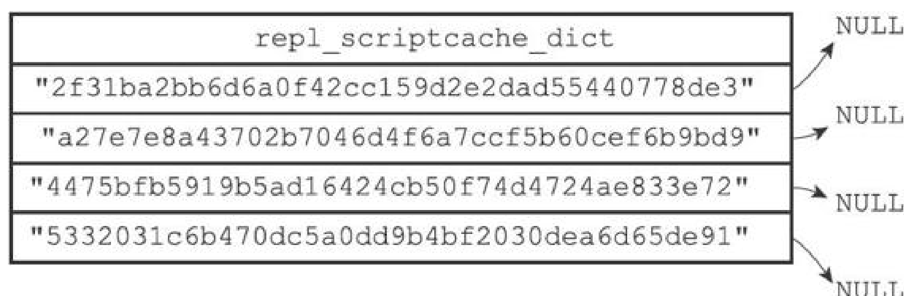


图20-14　执行EVALSHA命令之后的repl_scriptcache_dict字典

20.7 重点回顾

·Redis服务器在启动时，会对内嵌的Lua环境执行一系列修改操作，从而确保内嵌的Lua环境可以满足Redis在功能性、安全性等方面的需要。

·Redis服务器专门使用一个伪客户端来执行Lua脚本中包含的Redis命令。

·Redis使用脚本字典来保存所有被EVAL命令执行过，或者被SCRIPT LOAD命令载入过的Lua脚本，这些脚本可以用于实现SCRIPT EXISTS命令，以及实现脚本复制功能。

·EVAL命令为客户端输入的脚本在Lua环境中定义一个函数，并通过调用这个函数来执行脚本。

·EVALSHA命令通过直接调用Lua环境中已定义的函数来执行脚本。

·SCRIPT FLUSH命令会清空服务器lua_scripts字典中保存的脚本，并重置Lua环境。

·SCRIPT EXISTS命令接受一个或多个SHA1校验和为参数，并通过检查lua_scripts字典来确认校验和对应的脚本是否存在。

·SCRIPT LOAD命令接受一个Lua脚本为参数，为该脚本在Lua环境中创建函数，并将脚本保存到lua_scripts字典中。

·服务器在执行脚本之前，会为Lua环境设置一个超时处理钩子，当脚本出现超时运行情况时，客户端可以通过向服务器发送SCRIPT KILL命令来让钩子停止正在执行的脚本，或者发送SHUTDOWN nosave命令来让钩子关闭整个服务器。

·主服务器复制EVAL、SCRIPT FLUSH、SCRIPT LOAD三个命令的方法和复制普通Redis命令一样，只要将相同的命令传播给从服务器就可以了。

·主服务器在复制EVALSHA命令时，必须确保所有从服务器都已经载入了EVALSHA命令指定的SHA1校验和所对应的Lua脚本，如果不能确保这一点的话，主服务器会将EVALSHA命令转换成等效的EVAL命令，并通过传播EVAL命令来获得相同的脚本执行效果。

20.8 参考资料

《Lua 5.1 Reference Manual》对Lua语言的语法和标准库进行了很好的介绍：http://www.lua.org/manual/5.1/manual.html

第21章　排序

Redis的SORT命令可以对列表键、集合键或者有序集合键的值进行排序。

以下代码展示了SORT命令对列表键进行排序的例子：

```
redis> RPUSH numbers 5 3 1 4 2
(integer) 5
#
按插入顺序排列的列表元素
redis> LRANGE numbers 0 -1
1) "5"
2) "3"
3) "1"
4) "4"
5) "2"
#
按值从小到大有序排列的列表元素
redis> SORT numbers
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

以下代码展示了SORT命令使用ALPHA选项，对一个包含字符串值的集合键进行排序的例子：

```
redis> SADD alphabet a b c d e f g
(integer) 7
#
乱序排列的集合元素
redis> SMEMBERS alphabet
1) "d"
2) "a"
3) "f"
4) "e"
5) "b"
6) "g"
7) "c"
#
排序后的集合元素
redis> SORT alphabet ALPHA
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
7) "g"
```

接下来的例子使用了SORT命令和BY选项，以jack_number、peter_number、tom_number三个键的值为权重（weight），对有序集合test-result中的"jack"、"peter"、"tom"三个成员（member）进行排序：

```
redis> ZADD test-result 3.0 jack 3.5 peter 4.0 tom
```

```
(integer) 3
#
按元素的分值排列
redis> ZRANGE test-result 0 -1
1) "jack"
2) "peter"
3) "tom"
#
为各个元素设置序号
redis> MSET peter_number 1 tom_number 2 jack_number 3
OK
#
以序号为权重，对有序集合中的元素进行排序
redis> SORT test-result BY *_number
1) "peter"
2) "tom"
3) "jack"
```

本章将对SORT命令的实现原理进行介绍，并说明包括ASC、DESC、ALPHA、LIMIT、STORE、BY、GET在内的所有SORT命令选项的实现原理。

除此之外，本章还将说明当SORT命令同时使用多个选项时，各个不同选项的执行顺序，以及选项的执行顺序对排序结果所产生的影响。

21.1　SORT <key>命令的实现

SORT命令的最简单执行形式为：

```
SORT <key>
```

这个命令可以对一个包含数字值的键key进行排序。

以下示例展示了如何使用SORT命令对一个包含三个数字值的列表键进行排序：

```
redis> RPUSH numbers 3 1 2
(integer) 3
redis> SORT numbers
1) "1"
2) "2"
3) "3"
```

服务器执行SORT numbers命令的详细步骤如下：

1）创建一个和numbers列表长度相同的数组，该数组的每个项都是一个redis.h/redisSortObject结构，如图21-1所示。

array	
array[0] redisSortObject	obj
	u
array[1] redisSortObject	obj
	u
array[2] redisSortObject	obj
	u

图21-1　命令为排序numbers列表而创建的数组

2）遍历数组，将各个数组项的obj指针分别指向numbers列表的各个项，构成obj指针和列表项之间的一对一关系，如图21-2所示。

3）遍历数组，将各个obj指针所指向的列表项转换成一个double类型的浮点数，并将这个浮点数保存在相应数组项的u.score属性里面，如图21-3所示。

4）根据数组项u.score属性的值，对数组进行数字值排序，排序后的数组项按u.score属性的值从小到大排列，如图21-4所示。

5）遍历数组，将各个数组项的obj指针所指向的列表项作为排序结果返回给客户端，程序首先访问数组的索引0，返回u.score值为1.0的列表项"1"；然后访问数组的索引1，返回u.score值为2.0的列表项"2"；最后访问数组的索引2，返回u.score值为3.0的列表项"3"。

其他SORT＜key＞命令的执行步骤也和这里给出的SORT　numbers命令的执行步骤类似。

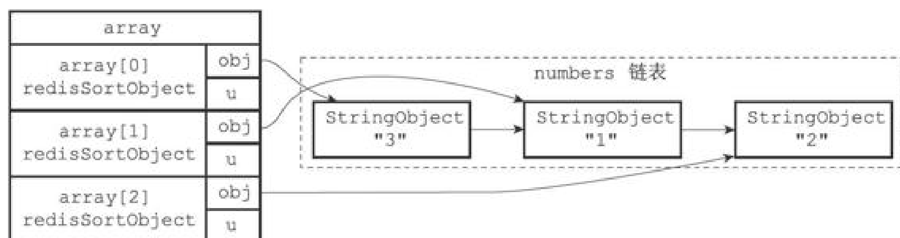


图21-2　将obj指针指向列表的各个项

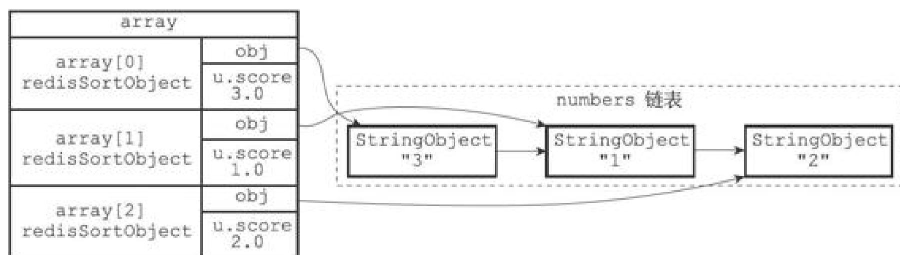


图21-3　设置数组项的u.score属性

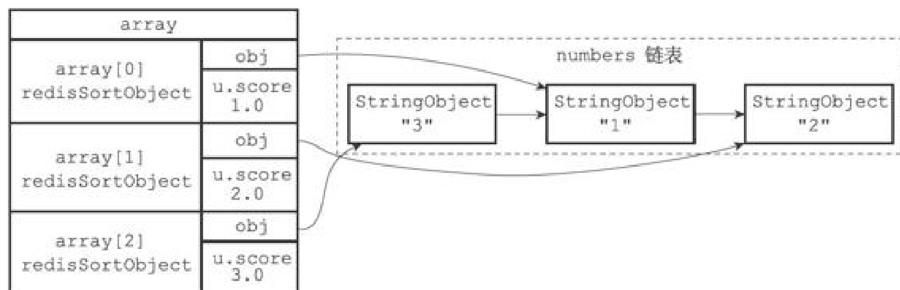


图21-4　排序后的数组

以下是redisSortObject结构的完整定义：

```
typedef struct _redisSortObject {
    //
被排序键的值
    robj *obj;
    //
权重
    union {
        //
排序数字值时使用
        double score;
        //
排序带有BY
选项的字符串值时使用
        robj *cmpobj;
    } u;
} redisSortObject;
```

SORT命令为每个被排序的键都创建一个与键长度相同的数组，数组的每个项都是一个redisSortObject结构，根据SORT命令使用的选项不同，程序使用redisSortObject结构的方式也不同，稍后介绍SORT命令的各种选项时我们会看到这一点。

21.2 ALPHA选项的实现

通过使用ALPHA选项，SORT命令可以对包含字符串值的键进行排序：

```
SORT <key> ALPHA
```

以下命令展示了如何使用SORT命令对一个包含三个字符串值的集合键进行排序：

```
redis> SADD fruits apple banana cherry
(integer) 3
#
元素在集合中是乱序存放的
redis> SMEMBERS fruits
1) "apple"
2) "cherry"
3) "banana"
#
对fruits
键进行字符串排序
redis> SORT fruits ALPHA
1) "apple"
2) "banana"
3) "cherry"
```

服务器执行SORT fruits ALPHA命令的详细步骤如下：

1）创建一个redisSortObject结构数组，数组的长度等于fruits集合的大小。

2）遍历数组，将各个数组项的obj指针分别指向fruits集合的各个元素，如图21-5所示。

3）根据obj指针所指向的集合元素，对数组进行字符串排序，排序后的数组项按集合元素的字符串值从小到大排列：因为"apple"、"banana"、"cherry"三个字符串的大小顺序为"apple"＜"banana"＜"cherry"，所以排序后数组的第一项指向"apple"元素，第二项指向"banana"元素，第三项指向"cherry"元素，如图21-6所示。

4）遍历数组，依次将数组项的obj指针所指向的元素返回给客户端。

其他SORT＜key＞ALPHA命令的执行步骤也和这里给出的SORT fruits ALPHA命令的执行步骤类似。

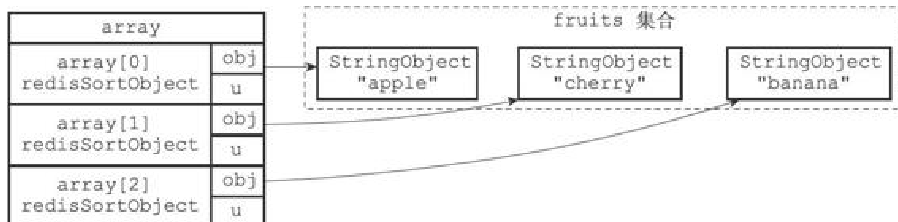


图21-5　将obj指针指向集合的各个元素

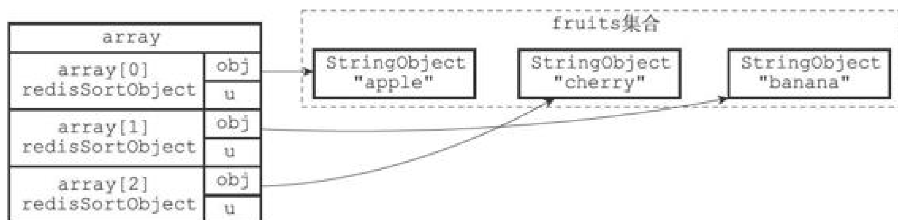


图21-6　按集合元素进行排序后的数组

21.3 ASC选项和DESC选项的实现

在默认情况下，SORT命令执行升序排序，排序后的结果按值的大小从小到大排列，以下两个命令是完全等价的：

```
SORT <key>
SORT <key> ASC
```

相反地，在执行SORT命令时使用DESC选项，可以让命令执行降序排序，让排序后的结果按值的大小从大到小排列：

```
SORT <key> DESC
```

以下是两个对numbers列表进行升序排序的例子，第一个命令根据默认设置，对numbers列表进行升序排序，而第二个命令则通过显式地使用ASC选项，对numbers列表进行升序排序，两个命令产生的结果完全一样：

```
redis> RPUSH numbers 3 1 2
(integer) 3
redis> SORT numbers
1) "1"
2) "2"
3) "3"
redis> SORT numbers ASC
1) "1"
2) "2"
3) "3"
```

与升序排序相反，以下是一个对numbers列表进行降序排序的例子：

```
redis> SORT numbers DESC
1) "3"
2) "2"
3) "1"
```

升序排序和降序排序都由相同的快速排序算法执行，它们之间的不同之处在于：

·在执行升序排序时，排序算法使用的对比函数产生升序对比结果。

·而在执行降序排序时，排序算法所使用的对比函数产生降序对比结果。

因为升序对比和降序对比的结果正好相反，所以它们会产生元素排列方式正好相反的两种排序结果。以numbers列表为例：

·图21-7展示了SORT命令在对numbers列表执行升序排序时所创建

的数组。

·图21-8展示了SORT命令在对numbers列表执行降序排序时所创建的数组。

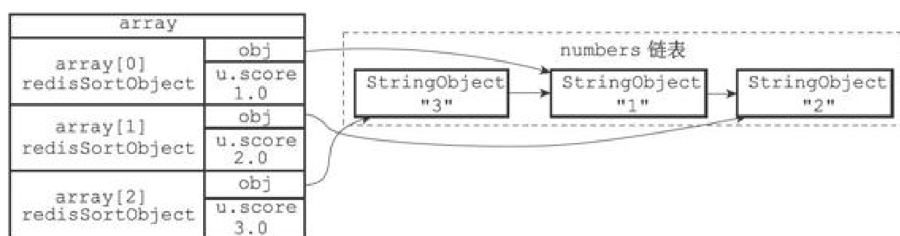


图21-7　执行升序排序的数组

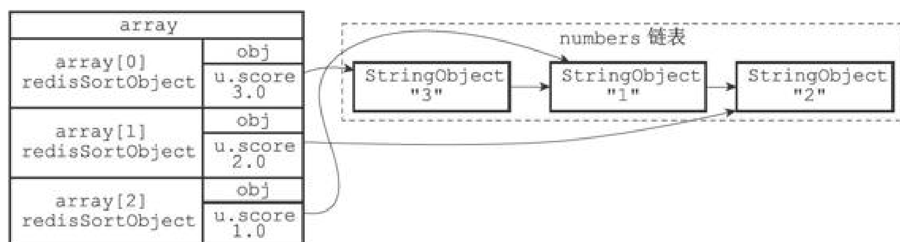


图21-8　执行降序排序的数组

其他SORT　<Key>　DESC命令的执行步骤也和这里给出的步骤类似。

21.4 BY选项的实现

在默认情况下，SORT命令使用被排序键包含的元素作为排序的权重，元素本身决定了元素在排序之后所处的位置。

例如，在下面这个例子里面，排序fruits集合所使用的权重就是"apple"、"banana"、"cherry"三个元素本身：

```
redis> SADD fruits "apple" "banana" "cherry"
(integer) 3
redis> SORT fruits ALPHA
1) "apple"
2) "banana"
3) "cherry"
```

另一方面，通过使用BY选项，SORT命令可以指定某些字符串键，或者某个哈希键所包含的某些域（field）来作为元素的权重，对一个键进行排序。

例如，以下这个例子就使用苹果、香蕉、樱桃三种水果的价钱，对集合键fruits进行了排序：

```
redis> MSET apple-price 8 banana-price 5.5 cherry-price 7
OK
redis> SORT fruits BY *-price
1) "banana"
2) "cherry"
3) "apple"
```

服务器执行SORT fruits BY*-price命令的详细步骤如下：

1）创建一个redisSortObject结构数组，数组的长度等于fruits集合的大小。

2）遍历数组，将各个数组项的obj指针分别指向fruits集合的各个元素，如图21-9所示。

3）遍历数组，根据各个数组项的obj指针所指向的集合元素，以及BY选项所给定的模式*-price，查找相应的权重键：

·对于"apple"元素，查找程序返回权重键"apple-price"。

·对于"banana"元素，查找程序返回权重键"banana-price"。

·对于"cherry"元素，查找程序返回权重键"cherry-price"。

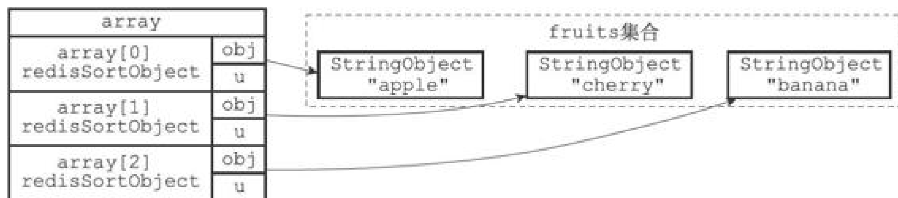


图21-9　将obj指针指向集合的各个元素

4）将各个权重键的值转换成一个double类型的浮点数，然后保存在相应数组项的u.score属性里面，如图21-10所示：

·"apple"元素的权重键"apple-price"的值转换之后为8.0。

·"banana"元素的权重键"banana-price"的值转换之后为5.5。

·"cherry"元素的权重键"cherry-price"的值转换之后为7.0。

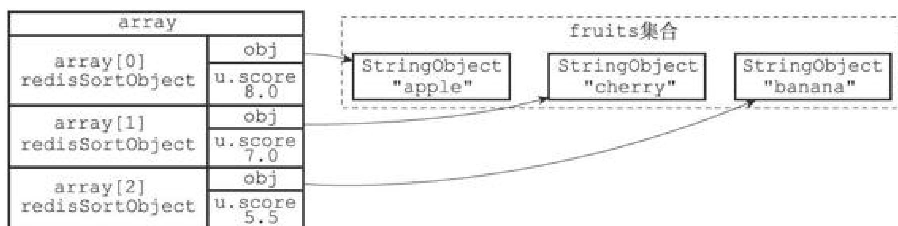


图21-10　根据权重键的值设置数组项的u.score属性

5）以数组项u.score属性的值为权重，对数组进行排序，得到一个按u.score属性的值从小到大排序的数组，如图21-11所示：

·权重为5.5的"banana"元素位于数组的索引0位置上。

·权重为7.0的"cherry"元素位于数组的索引1位置上。

·权重为8.0的"apple"元素位于数组的索引2位置上。

6）遍历数组，依次将数组项的obj指针所指向的集合元素返回给客户端。

其他SORT＜key＞BY＜pattern＞命令的执行步骤也和这里给出的步骤类似。

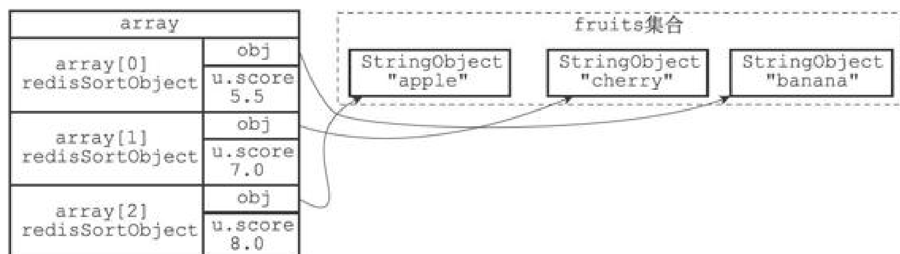


图21-11　根据u.score属性进行排序之后的数组

21.5 带有ALPHA选项的BY选项的实现

BY选项默认假设权重键保存的值为数字值，如果权重键保存的是字符串值的话，那么就需要在使用BY选项的同时，配合使用ALPHA选项。

举个例子，如果fruits集合包含的三种水果都有一个相应的字符串编号：

```
redis> SADD fruits "apple" "banana" "cherry"
(integer) 3
redis> MSET apple-id "FRUIT-25" banana-id "FRUIT-79" cherry-id "FRUIT-13"
OK
```

那么我们可以使用水果的编号为权重，对fruits集合进行排序：

```
redis> SORT fruits BY *-id ALPHA
1)"cherry"
2)"apple"
3)"banana"
```

服务器执行SORT fruits BY*-id ALPHA命令的详细步骤如下：

1）创建一个redisSortObject结构数组，数组的长度等于fruits集合的大小。

2）遍历数组，将各个数组项的obj指针分别指向fruits集合的各个元素，如图21-12所示。

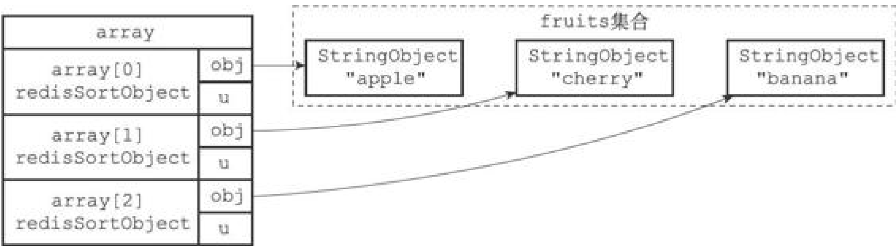


图21-12 将obj指针指向集合的各个元素

3）遍历数组，根据各个数组项的obj指针所指向的集合元素，以及BY选项所给定的模式*-id，查找相应的权重键：

·对于"apple"元素，查找程序返回权重键"apple-id"。

·对于"banana"元素，查找程序返回权重键"banana-id"。

·对于"cherry"元素，查找程序返回权重键"cherry-id"。

4）将各个数组项的u.cmpobj指针分别指向相应的权重键（一个字

符串对象），如图21-13所示。

5）以各个数组项的权重键的值为权重，对数组执行字符串排序，结果如图12-14所示：

·权重为"FRUIT-13"的"cherry"元素位于数组的索引0位置上。

·权重为"FRUIT-25"的"apple"元素位于数组的索引1位置上。

·权重为"FRUIT-79"的"banana"元素位于数组的索引2位置上。

6）遍历数组，依次将数组项的obj指针所指向的集合元素返回给客户端。

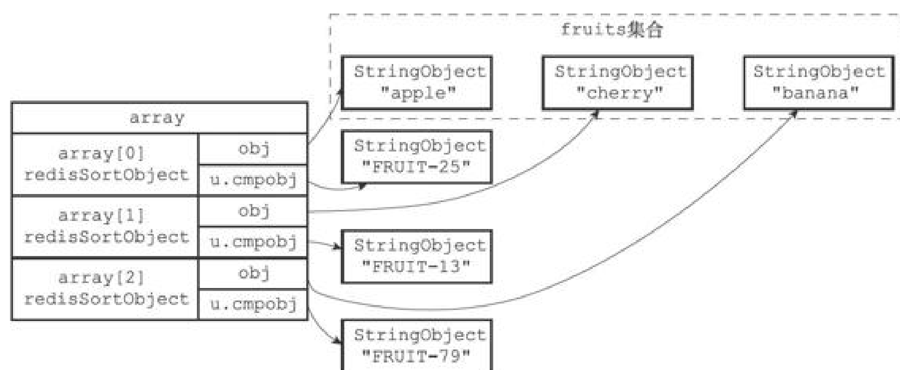


图21-13　将u.cmpobj指针指向权重键

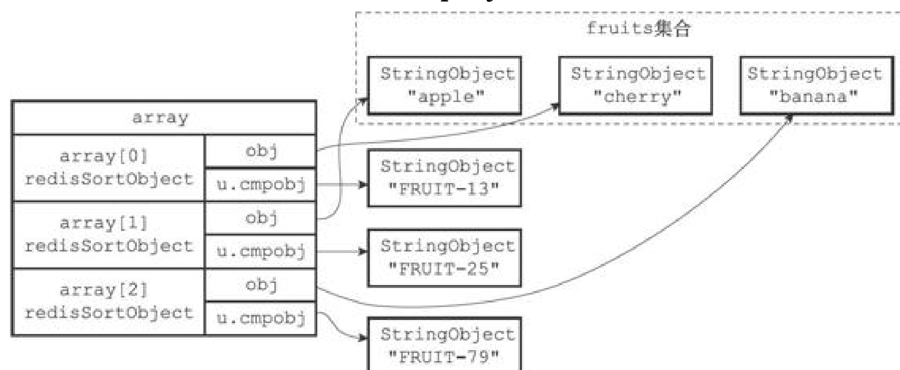


图21-14　按u.cmpobj所指向的字符串对象进行排序之后的数组

其他SORT　＜key＞　BY　＜pattern＞ALPHA命令的执行步骤也和这里给出的步骤类似。

21.6 LIMIT选项的实现

在默认情况下，SORT命令总会将排序后的所有元素都返回给客户端：

```
redis> SADD alphabet a b c d e f
(integer) 6
#
集合中的元素是乱序存放的
redis> SMEMBERS alphabet
1) "d"
2) "c"
3) "a"
4) "b"
5) "f"
6) "e"
#
对集合进行排序，并返回所有排序后的元素
redis> SORT alphabet ALPHA
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
```

但是，通过LIMIT选项，我们可以让SORT命令只返回其中一部分已排序的元素。

LIMIT选项的格式为LIMIT＜offset＞＜count＞：

·offset参数表示要跳过的已排序元素数量。

·count参数表示跳过给定数量的已排序元素之后，要返回的已排序元素数量。

举个例子，以下代码首先对alphabet集合进行排序，接着跳过0个已排序元素，然后返回4个已排序元素：

```
redis> SORT alphabet ALPHA LIMIT 0 4
1) "a"
2) "b"
3) "c"
4) "d"
```

与此类似，以下代码首先对alphabet集合进行排序，接着跳过2个已排序元素，然后返回3个已排序元素：

```
redis> SORT alphabet ALPHA LIMIT 2 3
1) "c"
2) "d"
3) "e"
```

服务器执行SORT alphabet ALPHA LIMIT 0 4命令的详细步骤如

下：

1）创建一个redisSortObject结构数组，数组的长度等于alphabet集合的大小。

2）遍历数组，将各个数组项的obj指针分别指向alphabet集合的各个元素，如图21-15所示。

3）根据obj指针所指向的集合元素，对数组进行字符串排序，排序后的数组如图21-16所示。

4）根据选项LIMIT 0 4，将指针移动到数组的索引0上面，然后依次访问array[0]、array[1]、array[2]、array[3]这4个数组项，并将数组项的obj指针所指向的元素"a"、"b"、"c"、"d"返回给客户端。

服务器执行SORT alphabet ALPHA LIMIT 2 3命令时的第一至第三步都和执行SORT alphabet ALPHA LIMIT 0 4命令时的步骤一样，只是第四步有所不同，上面的第4步如下：

4）根据选项LIMIT 2 3，将指针移动到数组的索引2上面，然后依次访问array[2]、array[3]、array[4]这3个数组项，并将数组项的obj指针所指向的元素"c"、"d"、"e"返回给客户端。

SORT命令在执行其他带有LIMIT选项的排序操作时，执行的步骤也和这里给出的步骤类似。

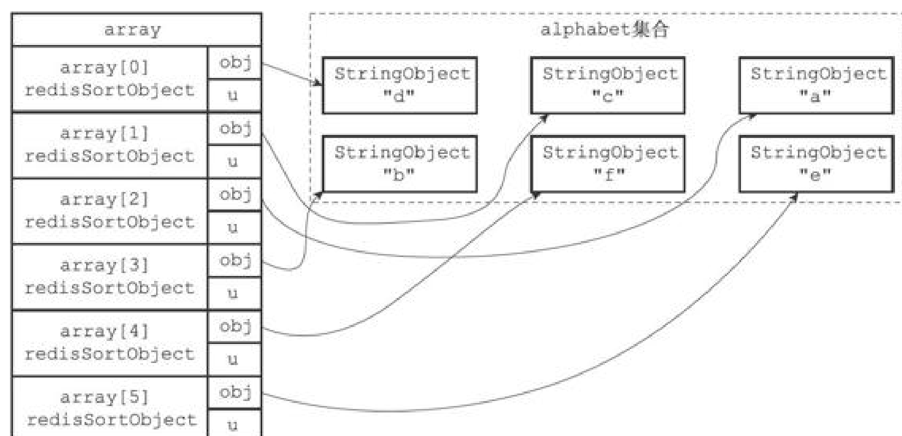


图21-15 将obj指针指向集合的各个元素

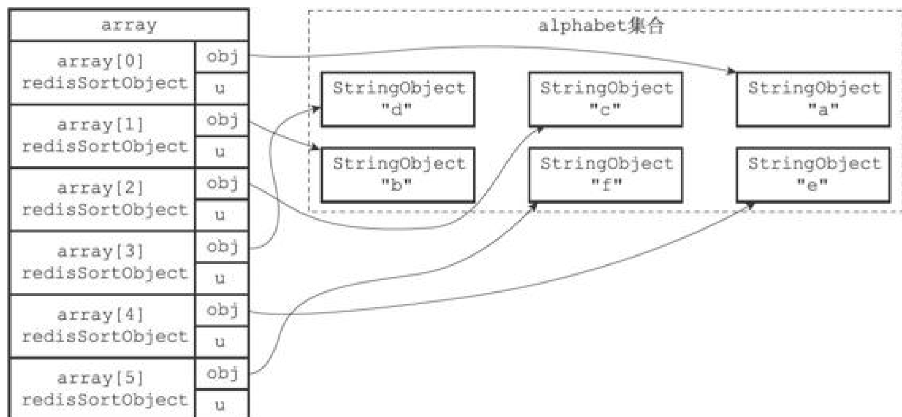


图21-16　排序后的数组

21.7 GET选项的实现

在默认情况下，SORT命令在对键进行排序之后，总是返回被排序键本身所包含的元素。

比如说，在以下这个对students集合进行排序的例子中，SORT命令返回的就是被排序之后的students集合的元素：

```
redis> SADD students "peter" "jack" "tom"
(integer) 3
redis> SORT students ALPHA
1) "jack"
2) "peter"
3) "tom"
```

但是，通过使用GET选项，我们可以让SORT命令在对键进行排序之后，根据被排序的元素，以及GET选项所指定的模式，查找并返回某些键的值。

比如说，在以下这个例子中，SORT命令首先对students集合进行排序，然后根据排序结果中的元素（学生的简称），查找并返回这些学生的全名：

```
#
设置peter
、jack
、tom
的全名
redis> SET peter-name "Peter White"
OK
redis> SET jack-name "Jack Snow"
OK
redis> SET tom-name "Tom Smith"
OK
# SORT
命令首先对students
集合进行排序，得到排序结果
# 1) "jack"
# 2) "peter"
# 3) "tom"
#
然后根据这些结果，获取并返回键jack-name
、peter-name
和tom-name
的值
redis> SORT students ALPHA GET *-name
1) "Jack Snow"
2) "Peter White"
3) "Tom Smith"
```

服务器执行SORT students ALPHA GET*-name命令的详细步骤如下：

1）创建一个redisSortObject结构数组，数组的长度等于students集合的大小。

2）遍历数组，将各个数组项的obj指针分别指向students集合的各个元素，如图21-17所示。

3）根据obj指针所指向的集合元素，对数组进行字符串排序，排序后的数组如图21-18所示：

·被排序到数组索引0位置的是"jack"元素。

·被排序到数组索引1位置的是"peter"元素。

·被排序到数组索引2位置的是"tom"元素。

4）遍历数组，根据数组项obj指针所指向的集合元素，以及GET选项所给定的*-name模式，查找相应的键：

·对于"jack"元素和*-name模式，查找程序返回键jack-name。

·对于"peter"元素和*-name模式，查找程序返回键peter-name。

·对于"tom"元素和*-name模式，查找程序返回键tom-name。

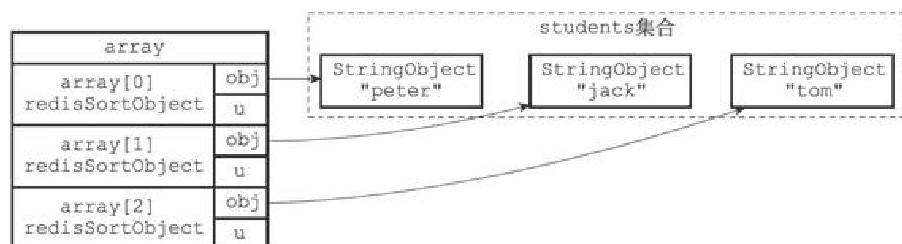


图21-17　排序之前的数组

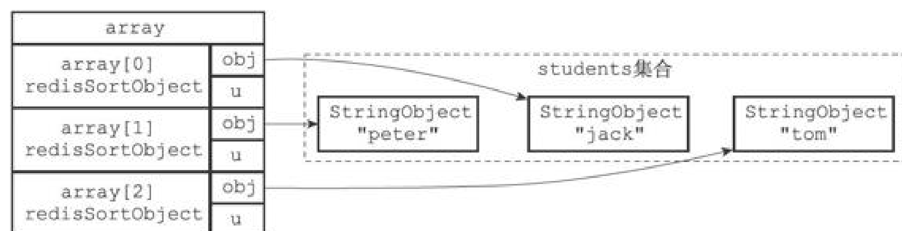


图21-18　排序之后的数组

5）遍历查找程序返回的三个键，并向客户端返回它们的值：

·首先返回的是jack-name键的值"Jack Snow"。

·然后返回的是peter-name键的值"Peter White"。

·最后返回的是tom-name键的值"Tom Smith"。

因为一个SORT命令可以带有多个GET选项，所以随着GET选项的增

多，命令要执行的查找操作也会增多。

举个例子，以下SORT命令对students集合进行了排序，并通过两个GET选项来获取被排序元素（一个学生）所对应的全名和出生日期：

```
#
为学生设置出生日期
redis> SET peter-birth 1995-6-7
OK
redis> SET tom-birth 1995-8-16
OK
redis> SET jack-birth 1995-5-24
OK
#
排序students
集合，并获取相应的全名和出生日期
redis> SORT students ALPHA GET *-name GET *-birth
1) "Jack Snow"
2) "1995-5-24"
3) "Peter White"
4) "1995-6-7"
5) "Tom Smith"
6) "1995-8-16"
```

服务器执行SORT students ALPHA GET*-name GET*-birth命令的前三个步骤，和执行SORT students ALPHA GET*-name命令时的前三个步骤相同，但从第四步开始有所区别：

4）遍历数组，根据数组项obj指针所指向的集合元素，以及两个GET选项所给定的*-name模式和*-birth模式，查找相应的键：

·对于"jack"元素和*-name模式，查找程序返回jack-name键。

·对于"jack"元素和*-birth模式，查找程序返回jack-birth键。

·对于"peter"元素和*-name模式，查找程序返回peter-name键。

·对于"peter"元素和*-birth模式，查找程序返回peter-birth键。

·对于"tom"元素和*-name模式，查找程序返回tom-name键。

·对于"tom"元素和*-birth模式，查找程序返回tom-birth键。

5）遍历查找程序返回的六个键，并向客户端返回它们的值：

·首先返回jack-name键的值"Jack Snow"。

·其次返回jack-birth键的值"1995-5-24"。

·之后返回peter-name键的值"Peter White"。

·再之后返回peter-birth键的值"1995-6-7"。

·然后返回tom-name键的值"Tom Smith"。

·最后返回tom-birth键的值"1995-8-16"。

SORT命令在执行其他带有GET选项的排序操作时，执行的步骤也和这里给出的步骤类似。

21.8　STORE选项的实现

在默认情况下，SORT命令只向客户端返回排序结果，而不保存排序结果：

```
redis> SADD students "peter" "jack" "tom"
(integer) 3
redis> SORT students ALPHA
1) "jack"
2) "peter"
3) "tom"
```

但是，通过使用STORE选项，我们可以将排序结果保存在指定的键里面，并在有需要时重用这个排序结果：

```
redis> SORT students ALPHA STORE sorted_students
(integer) 3
redis> LRANGE sorted_students 0-1
1) "jack"
2) "peter"
3) "tom"
```

服务器执行SORT students ALPHA STORE sorted_students命令的详细步骤如下：

1）创建一个redisSortObject结构数组，数组的长度等于students集合的大小。

2）遍历数组，将各个数组项的obj指针分别指向students集合的各个元素。

3）根据obj指针所指向的集合元素，对数组进行字符串排序，排序后的数组如图21-19所示：

·被排序到数组索引0位置的是"jack"元素。

·被排序到数组索引1位置的是"peter"元素。

·被排序到数组索引2位置的是"tom"元素。

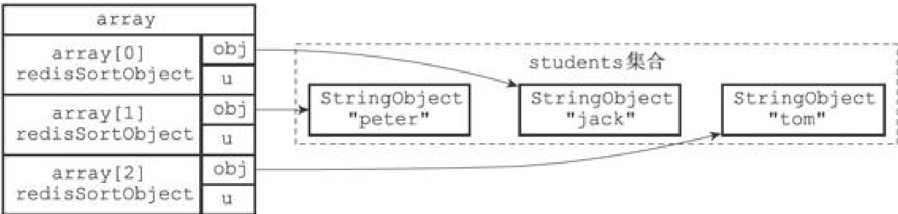


图21-19　排序之后的数组

4）检查sorted_students键是否存在，如果存在的话，那么删除该键。

5）设置sorted_students为空白的列表键。

6）遍历数组，将排序后的三个元素"jack"、"peter"和"tom"依次推入sorted_students列表的末尾，相当于执行命令RPUSH sorted_students"jack"、"peter"、"tom"。

7）遍历数组，向客户端返回"jack"、"peter"、"tom"三个元素。

SORT命令在执行其他带有STORE选项的排序操作时，执行的步骤也和这里给出的步骤类似。

21.9 多个选项的执行顺序

前面的章节介绍了SORT命令以及相关选项的实现原理，为了简单起见，在介绍单个选项的实现原理时，文章通常只在代码示例中使用被介绍的那个选项，但在SORT命令的实际使用中，情况并不总是那么简单的，一个SORT命令请求通常会用到多个选项，而这些选项的执行顺序是有先后之分的。

21.9.1 选项的执行顺序

如果按照选项来划分的话，一个SORT命令的执行过程可以分为以下四步：

1）排序：在这一步，命令会使用ALPHA、ASC或DESC、BY这几个选项，对输入键进行排序，并得到一个排序结果集。

2）限制排序结果集的长度：在这一步，命令会使用LIMIT选项，对排序结果集的长度进行限制，只有LIMIT选项指定的那部分元素会被保留在排序结果集中。

3）获取外部键：在这一步，命令会使用GET选项，根据排序结果集中的元素，以及GET选项指定的模式，查找并获取指定键的值，并用这些值来作为新的排序结果集。

4）保存排序结果集：在这一步，命令会使用STORE选项，将排序结果集保存到指定的键上面去。

5）向客户端返回排序结果集：在最后这一步，命令遍历排序结果集，并依次向客户端返回排序结果集中的元素。

在以上这些步骤中，后一个步骤必须在前一个步骤完成之后进行。

举个例子，如果客户端向服务器发送以下命令：

```
SORT <key> ALPHA DESC BY <by-pattern> LIMIT <offset> <count> GET <get-pattern> STORE <store_key>
```

那么命令首先会执行：

```
SORT <key> ALPHA DESC BY <by-pattern>
```

接着执行：

```
LIMIT <offset> <count>
```

然后执行：

```
GET <get-pattern>
```

之后执行：

```
STORE <store_key>
```

最后，命令遍历排序结果集，将结果集中的元素依次返回给客户端。

21.9.2 选项的摆放顺序

另外要提醒的一点是，调用SORT命令时，除了GET选项之外，改变选项的摆放顺序并不会影响SORT命令执行这些选项的顺序。

例如，命令：

```
SORT <key> ALPHA DESC BY <by-pattern> LIMIT <offset> <count> GET <get-pattern> STORE <store_key>
```

和命令：

```
SORT <key> LIMIT <offset> <count> BY <by-pattern> ALPHA GET <get-pattern> STORE <store_key> DESC
```

以及命令：

```
SORT <key> STORE <store_key> DESC BY <by-pattern> GET <get-pattern> ALPHA LIMIT <offset> <count>
```

都产生完全相同的排序数据集。

不过，如果命令包含了多个GET选项，那么在调整选项的位置时，我们必须保证多个GET选项的摆放顺序不变，这才可以让排序结果集保持不变。

例如，命令：

```
SORT <key> GET <pattern-a> GET <pattern-b> STORE <store_key>
```

和命令：

```
SORT <key> STORE <store_key> GET <pattern-a> GET <pattern-b>
```

产生的排序结果集是完全一样的，但如果将两个GET选项的顺序调

整一下：

```
SORT <key> STORE <store_key> GET <pattern-b> GET <pattern-a>
```

那么这个命令产生的排序结果集就会和前面两个命令产生的排序结果集不同。

因此在调整SORT命令各个选项的摆放顺序时，必须小心处理GET选项。

21.10 重点回顾

·SORT命令通过将被排序键包含的元素载入到数组里面，然后对数组进行排序来完成对键进行排序的工作。

·在默认情况下，SORT命令假设被排序键包含的都是数字值，并且以数字值的方式来进行排序。

·如果SORT命令使用了ALPHA选项，那么SORT命令假设被排序键包含的都是字符串值，并且以字符串的方式来进行排序。

·SORT命令的排序操作由快速排序算法实现。

·SORT命令会根据用户是否使用了DESC选项来决定是使用升序对比还是降序对比来比较被排序的元素，升序对比会产生升序排序结果，被排序的元素按值的大小从小到大排列，降序对比会产生降序排序结果，被排序的元素按值的大小从大到小排列。

·当SORT命令使用了BY选项时，命令使用其他键的值作为权重来进行排序操作。

·当SORT命令使用了LIMIT选项时，命令只保留排序结果集中LIMIT选项指定的元素。

·当SORT命令使用了GET选项时，命令会根据排序结果集中的元素，以及GET选项给定的模式，查找并返回其他键的值，而不是返回被排序的元素。

·当SORT命令使用了STORE选项时，命令会将排序结果集保存在指定的键里面。

·当SORT命令同时使用多个选项时，命令先执行排序操作（可用的选项为ALPHA、ASC或DESC、BY），然后执行LIMIT选项，之后执行GET选项，再之后执行STORE选项，最后才将排序结果集返回给客户端。

·除了GET选项之外，调整选项的摆放位置不会影响SORT命令的排序结果。

第22章　二进制位数组

Redis提供了SETBIT、GETBIT、BITCOUNT、BITOP四个命令用于处理二进制位数组（bit array，又称“位数组”）。

其中，SETBIT命令用于为位数组指定偏移量上的二进制位设置值，位数组的偏移量从0开始计数，而二进制位的值则可以是0或者1：

```
redis> SETBIT bit 0 1  # 0000 0001
(integer) 0
redis> SETBIT bit 3 1        # 0000 1001
(integer) 0
redis> SETBIT bit 0 0        # 0000 1000
(integer) 1
```

而GETBIT命令则用于获取位数组指定偏移量上的二进制位的值：

```
redis> GETBIT bit 0 # 0000 1000
(integer) 0
redis> GETBIT bit 3 # 0000 1000
(integer) 1
```

BITCOUNT命令用于统计位数组里面，值为1的二进制位的数量：

```
redis> BITCOUNT bit  # 0000 1000
(integer) 1
redis> SETBIT bit 0 1        # 0000 1001
(integer) 0
redis> BITCOUNT bit
(integer) 2
redis> SETBIT bit 1 1        # 0000 1011
(integer) 0
redis> BITCOUNT bit
(integer) 3
```

最后，BITOP命令既可以对多个位数组进行按位与（and）、按位或（or）、按位异或（xor）运算：

```
redis> SETBIT x 3 1        # x = 0000 1011
(integer) 0
redis> SETBIT x 1 1
(integer) 0
redis> SETBIT x 0 1
(integer) 0
redis> SETBIT y 2 1  # y = 0000 0110
(integer) 0
redis> SETBIT y 1 1
(integer) 0
redis> SETBIT z 2 1  # z = 0000 0101
(integer) 0
redis> SETBIT z 0 1
(integer) 0
redis> BITOP AND and-result x y z    # 0000 0000
(integer) 1
redis> BITOP OR or-result x y z              # 0000 1111
(integer) 1
redis> BITOP XOR xor-result x y z    # 0000 1000
```

```
(integer) 1
```

也可以对给定的位数组进行取反（not）运算：

```
redis> SETBIT value 0 1                # 0000 1001
(integer) 0
redis> SETBIT value 3 1
(integer) 0
redis> BITOP NOT not-value value     # 1111 0110
(integer) 1
```

本章将对Redis表示位数组的方法进行说明，并介绍GETBIT、SETBIT、BITCOUNT、BITOP四个命令的实现原理。

22.1 位数组的表示

Redis使用字符串对象来表示位数组，因为字符串对象使用的SDS数据结构是二进制安全的，所以程序可以直接使用SDS结构来保存位数组，并使用SDS结构的操作函数来处理位数组。

图22-1展示了用SDS表示的，一字节长的位数组：

·redisObject.type的值为REDIS_STRING，表示这是一个字符串对象。

·sdshdr.len的值为1，表示这个SDS保存了一个一字节长的位数组。

·buf数组中的buf[0]字节保存了一字节长的位数组。

·buf数组中的buf[1]字节保存了SDS程序自动追加到值的末尾的空字符'\0'。

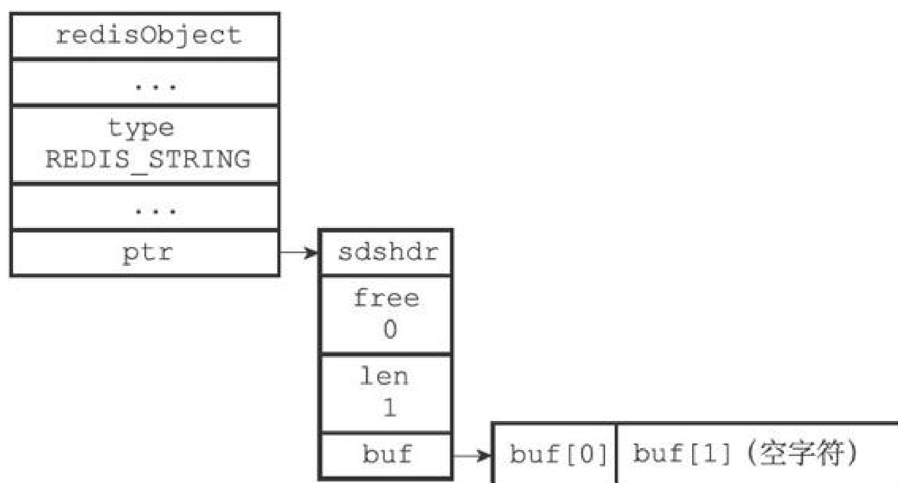


图22-1　SDS表示的位数组

因为本章介绍的操作涉及二进制位，为了清晰地展示各个位的值，本章会对SDS中buf数组的展示方式进行一些修改，让各个字节的各个位都可以清楚地展现出来。比如说，本章会将前面图22-1展示的SDS值改成图22-2所示的样子。

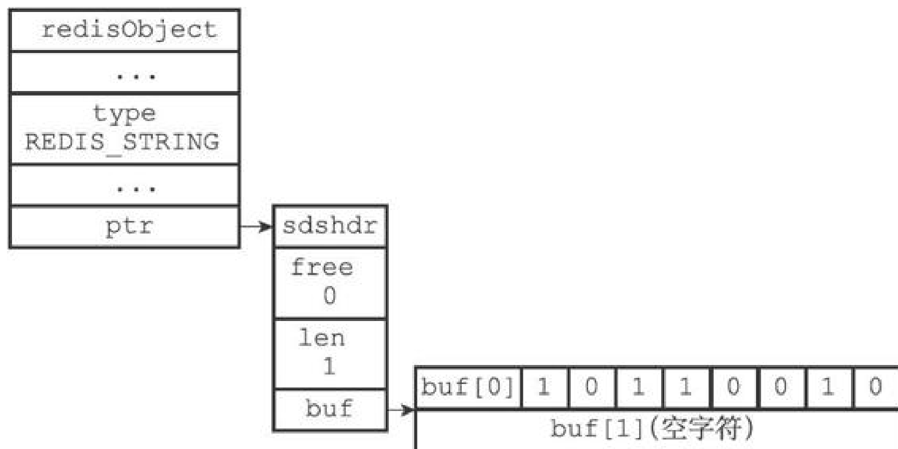


图22-2　一字节长的位数组的SDS表示

现在，buf数组的每个字节都用一行来表示，每行的第一个格子buf[i]表示这是buf数组的哪个字节，而buf[i]之后的八个格子则分别代表这一字节中的八个位。

需要注意的是，buf数组保存位数组的顺序和我们平时书写位数组的顺序是完全相反的，例如，在图22-2的buf[0]字节中，各个位的值分别是1、0、1、1、0、0、1、0，这表示buf[0]字节保存的位数组为0100 1101。使用逆序来保存位数组可以简化SETBIT命令的实现，详细的情况稍后在介绍SETBIT命令的实现原理时会说到。

图22-3展示了另一个位数组示例：

·sdshdr.len属性的值为3，表示这个SDS保存了一个三字节长的位数组。

·位数组由buf数组中的buf[0]、buf[1]、buf[2]三个字节保存，和之前说明的一样，buf数组使用逆序来保存位数组：位数组1111 0000 1100 0011 1010 0101在buf数组中会被保存为1010 0101 1100 0011 0000 1111。

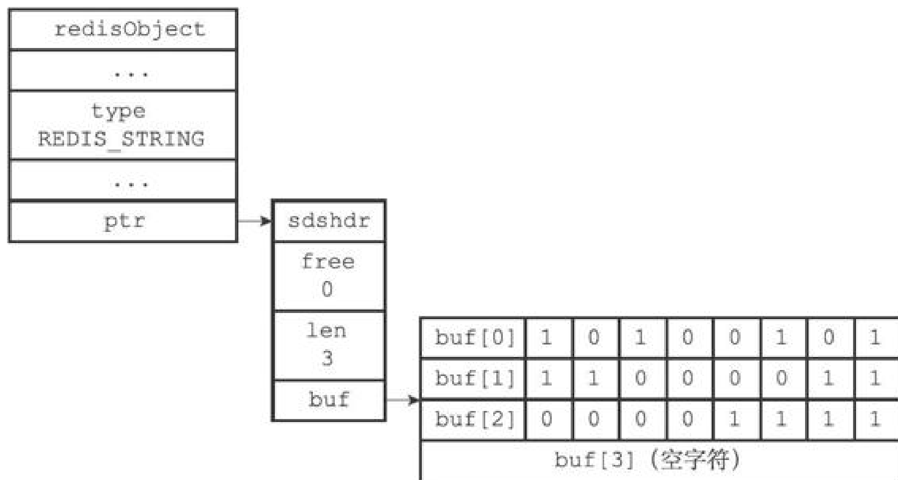


图22-3　三字节长的位数组的SDS表示

22.2　GETBIT命令的实现

GETBIT命令用于返回位数组bitarray在offset偏移量上的二进制位的值：

```
GETBIT <bitarray> <offset>
```

GETBIT命令的执行过程如下：

1）计算byte=⌊offset÷8⌋，byte值记录了offset偏移量指定的二进制位保存在位数组的哪个字节。

2）计算bit=（offset　mod　8）+1，bit值记录了offset偏移量指定的二进制位是byte字节的第几个二进制位。

3）根据byte值和bit值，在位数组bitarray中定位offset偏移量指定的二进制位，并返回这个位的值。

举个例子，对于图22-2所示的位数组来说，命令：

```
GETBIT <bitarray> 3
```

将执行以下操作：

1）⌊3÷8⌋的值为0。

2）（3 mod 8）+1的值为4。

3）定位到buf[0]字节上面，然后取出该字节上的第4个二进制位（从左向右数）的值。

4）向客户端返回二进制位的值1。

命令的执行过程如图22-4所示。

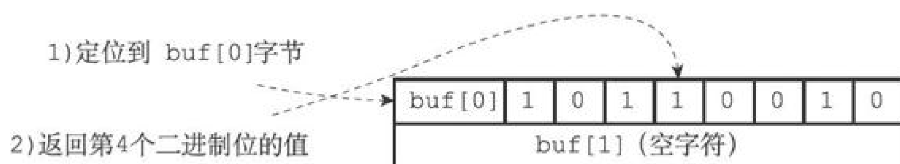
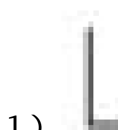


图22-4　查找并返回offset为3的二进制位的过程

再举一个例子，对于图22-3所示的位数组来说，命令：

```
GETBIT <bitarray> 10
```

将执行以下操作：

1）10÷8」的值为1。

2）（10 mod 8）+1的值为3。

3）定位到buf[1]字节上面，然后取出该字节上的第3个二进制位的值。

4）向客户端返回二进制位的值0。

命令的执行过程如图22-5所示。

1）定位到buf[1]字节

2）返回第3个二进制位的值

buf[0]	1	0	1	0	0	1	0	1
buf[1]	1	1	0	0	0	0	1	1
buf[2]	0	0	0	0	1	1	1	1
buf[3]	0	0	0	0	0	0	0	0

图22-5　查找并返回offset为10的二进制位的过程

因为GETBIT命令执行的所有操作都可以在常数时间内完成，所以该命令的算法复杂度为O（1）。

22.3　SETBIT命令的实现

SETBIT用于将位数组bitarray在offset偏移量上的二进制位的值设置为value，并向客户端返回二进制位被设置之前的旧值：

```
SETBIT <bitarray> <offset> <value>
```

以下是SETBIT命令的执行过程：

1）计算len＝⌊offset÷8⌋＋1，len值记录了保存offset偏移量指定的二进制位至少需要多少字节。

2）检查bitarray键保存的位数组（也即是SDS）的长度是否小于len，如果是的话，将SDS的长度扩展为len字节，并将所有新扩展空间的二进制位的值设置为0。

3）计算byte＝⌊offset÷8⌋，byte值记录了offset偏移量指定的二进制位保存在位数组的哪个字节。

4）计算bit＝（offset　mod　8）＋1，bit值记录了offset偏移量指定的二进制位是byte字节的第几个二进制位。

5）根据byte值和bit值，在bitarray键保存的位数组中定位offset偏移量指定的二进制位，首先将指定二进制位现在值保存在oldvalue变量，然后将新值value设置为这个二进制位的值。

6）向客户端返回oldvalue变量的值。

因为SETBIT命令执行的所有操作都可以在常数时间内完成，所以该命令的时间复杂度为O（1）。

22.3.1　SETBIT命令的执行示例

让我们通过观察一些SETBIT命令的执行例子来熟悉SETBIT命令的运行过程。

首先，如果我们对图22-2所示的位数组执行命令：

```
SETBIT <bitarray> 1 1
```

那么服务器将执行以下操作：

1）计算$\lfloor 1 \div 8 \rfloor + 1$，得出值1，这表示保存偏移量为1的二进制位至少需要1字节长位数组。

2）检查位数组的长度，发现SDS的长度不小于1字节，无须执行扩展操作。

3）计算$\lfloor 1 \div 8 \rfloor$，得出值0，说明偏移量为1的二进制位位于buf[0]字节。

4）计算（1 mod 8）+1，得出值2，说明偏移量为1的二进制位是buf[0]字节的第2个二进制位。

5）定位到buf[0]字节的第2个二进制位上面，将二进制位现在的值0保存到oldvalue变量，然后将二进制位的值设置为1。

6）向客户端返回oldvalue变量的值0。

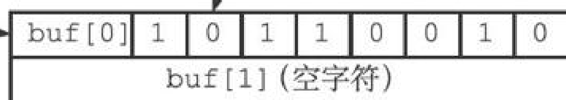


图22-6　SETBIT命令的执行过程

图22-6展示了SETBIT命令的执行过程，而图22-7则展示了SETBIT命令执行之后，位数组的样子。



图22-7　SETBIT命令执行之后的位数组

22.3.2　带扩展操作的SETBIT命令示例

前面展示的SETBIT例子无须对位数组进行扩展，现在，让我们来看一个需要对位数组进行扩展的例子。

假设我们对图22-2所示的位数组执行命令：

```
SETBIT <bitarray> 12 1
```

那么服务器将执行以下操作：

1）计算$\lfloor 12 \div 8 \rfloor + 1$，得出值2，这表示保存偏移量为12的二进制位至少需要2字节长的位数组。

2）对位数组的长度进行检查，得知位数组现在的长度为1字节，这比执行命令所需的最小长度2字节要小，所以程序会要求将位数组的长度扩展为2字节。不过，尽管程序只要求2字节长的位数组，但SDS的空间预分配策略会为SDS额外多分配2字节的未使用空间，再加上为保存空字符而额外分配的1字节，扩展之后buf数组的实际长度为5字节，如图22-8所示。

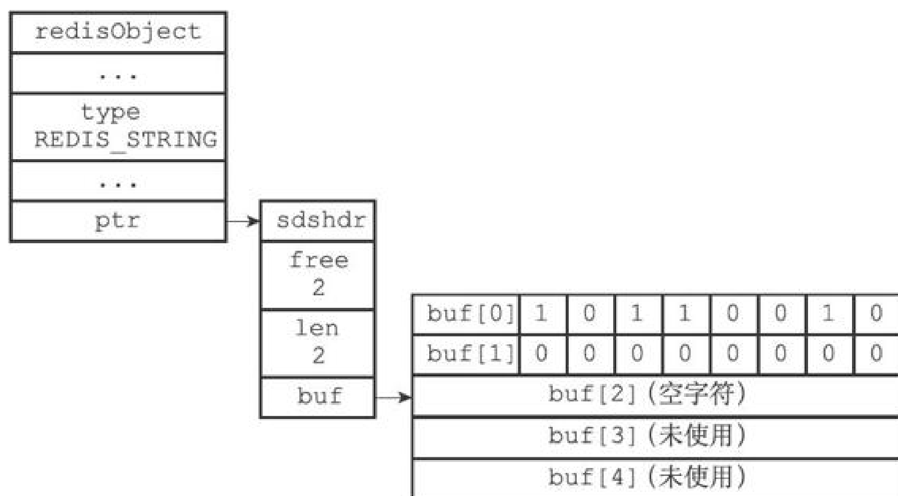


图22-8　扩展空间之后的位数组

3）计算$\lfloor 12 \div 8 \rfloor$，得出值1，说明偏移量为12的二进制位位于buf[1]字节中。

4）计算（12 mod 8）+1，得出值5，说明偏移量为12的二进制位是buf[1]字节的第5个二进制位。

5）定位到buf[1]字节的第5个二进制位，将二进制位现在的值0保存到oldvalue变量，然后将二进制位的值设置为1。

6）向客户端返回oldvalue变量的值0。

图22-9展示了SETBIT命令定位并设置指定二进制位的过程，而图22-10则展示了SETBIT命令执行之后，位数组的样子。

1）定位到buf[1]字节

2）定位到buf[1]字节的第5个二进制位
首先将位现在的值0保存到oldvalue
变量然后将位的值设置为1

buf[0]	1	0	1	1	0	0	1	0
buf[1]	0	0	0	0	0	0	0	0
buf[2]（空字符）								
buf[3]（未使用）								
buf[4]（未使用）								

图22-9　SETBIT命令的执行过程

buf[0]	1	0	1	1	0	0	1	0
buf[1]	0	0	0	0	1	0	0	0
buf[2]（空字符）								
buf[3]（未使用）								
buf[4]（未使用）								

图22-10　执行SETBIT命令之后的位数组

注意，因为buf数组使用逆序来保存位数组，所以当程序对buf数组进行扩展之后，写入操作可以直接在新扩展的二进制位中完成，而不必改动位数组原来已有的二进制位。

相反地，如果buf数组使用和书写位数组时一样的顺序来保存位数组，那么在每次扩展buf数组之后，程序都需要将位数组已有的位进行移动，然后才能执行写入操作，这比SETBIT命令目前的实现方式要复杂，并且移位带来的CPU时间消耗也会影响命令的执行速度。

图22-11至图22-14模拟了程序在buf数组按书写顺序保存位数组的情况下，对位数组0100 1101执行命令SETBIT ＜bitarray＞ 12 1，将值改为0001 0000 0100 1101的整个过程。

buf[0]	0	1	0	0	1	1	0	1
buf[1]（空字符）								

图22-11　按书写顺序保存的位数组0100 1101

buf[0]	0	1	0	0	1	1	0	1
buf[1]	0	0	0	0	0	0	0	0
buf[2]（空字符）								
buf[3]（未使用）								
buf[4]（未使用）								

图22-12　扩展之后的位数组

将字节buf[0]的所有二进制位移动到字节buf[1]

buf[0]	0	0	0	0	0	0	0	0
buf[1]	0	1	0	0	1	1	0	1
buf[2]（空字符）								
buf[3]（未使用）								
buf[4]（未使用）								

图22-13　移动已有的二进制位

将偏移量为12的二进制位的值设置为1

buf[0]	0	0	0	1	0	0	0	0
buf[1]	0	1	0	0	1	1	0	1
buf[2]（空字符）								
buf[3]（未使用）								
buf[4]（未使用）								

图22-14　设置指定二进制位的值

22.4 BITCOUNT命令的实现

BITCOUNT命令用于统计给定位数组中，值为1的二进制位的数量。

举个例子，对于图22-15所示的位数组来说，BITCOUNT命令将返回4。

而对于图22-16所示的位数组来说，BITCOUNT命令将返回12。

buf[0]	1	0	1	1	0	0	1	0
buf[1]（空字符）								

图22-15 BITCOUNT命令示例一

buf[0]	1	0	1	0	0	1	0	1
buf[1]	1	1	0	0	0	0	1	1
buf[2]	0	0	0	0	1	1	1	1
buf[3]（空字符）								

图22-16 BITCOUNT命令示例二

BITCOUNT命令要做的工作初看上去并不复杂，但实际上要高效地实现这个命令并不容易，需要用到一些精巧的算法。

接下来的几个小节将对BITCOUNT命令可能使用的几种算法进行介绍，并最终给出BITCOUNT命令的具体实现原理。

22.4.1 二进制位统计算法（1）：遍历算法

实现BITCOUNT命令最简单直接的方法，就是遍历位数组中的每个二进制位，并在遇到值为1的二进制位时，将计数器的值增一。

图22-17展示了程序使用遍历算法，对一个8位长的位数组进行遍历并计数的整个过程。

counter=1

buf[0]	1	0	1	1	0	0	1	0

buf[1]（空字符）

counter=1

buf[0]	1	0	1	1	0	0	1	0

buf[1]（空字符）

counter=2

buf[0]	1	0	1	1	0	0	1	0

buf[1]（空字符）

counter=3

buf[0]	1	0	1	1	0	0	1	0

buf[1]（空字符）

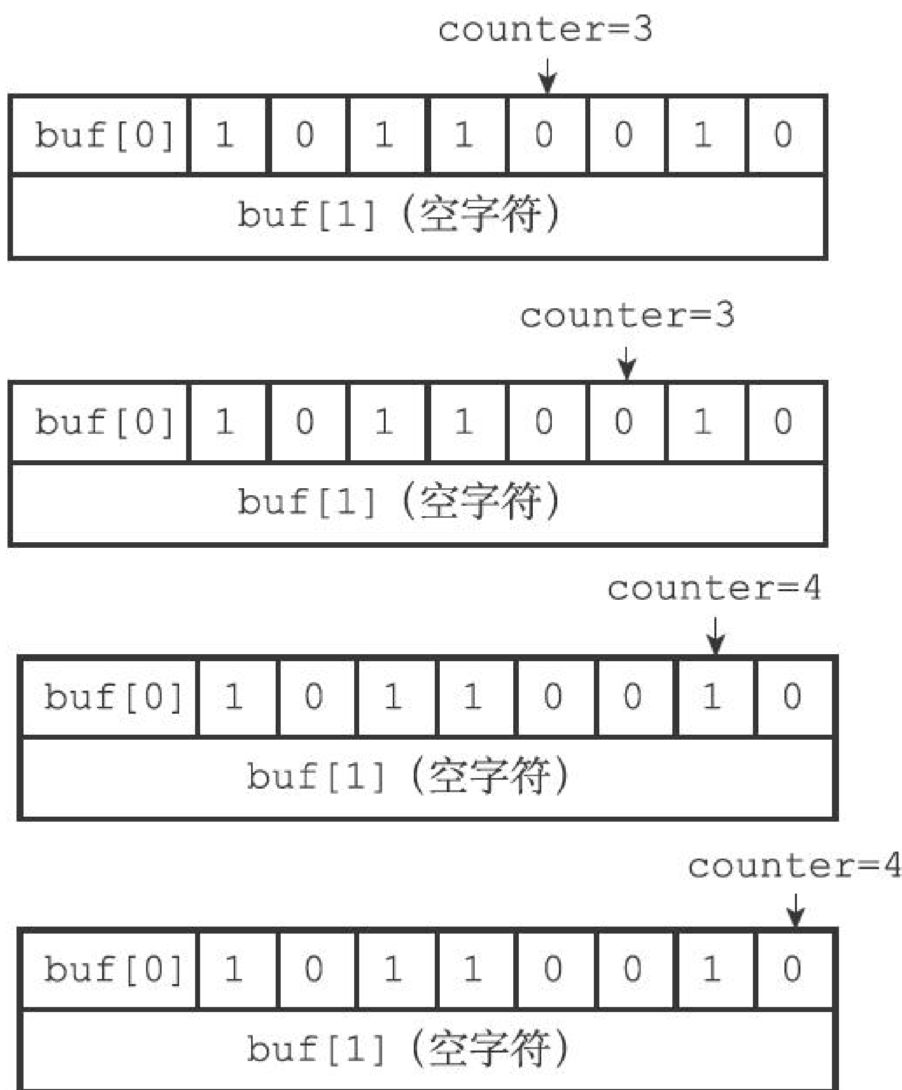


图22-17　遍历算法的运行过程

遍历算法虽然实现起来简单，但效率非常低，因为这个算法在每次循环中只能检查一个二进制位的值是否为1，所以检查操作执行的次数将与位数组包含的二进制位的数量成正比。

例如，假设要检查的位数组的长度为100MB，那么按1MB＝1000000Byte＝8000000bit来计算，使用遍历算法检查长度为100MB的位数组将需要执行检查操作八亿次（100*8000000）！而对于长度为500MB的位数组来说，遍历算法将需要执行检查操作四十亿次！

尽管遍历算法对单个二进制位的检查可以在很短的时间内完成，但重复执行上亿次这种检查肯定不是一个高效程序应有的表现，为了让BITCOUNT命令的实现尽可能地高效，程序必须尽可能地增加每次检查

所能处理的二进制位的数量，从而减少检查操作执行的次数。

22.4.2　二进制位统计算法（2）：查表算法

优化检查操作的一个办法是使用查表法：

·对于一个有限集合来说，集合元素的排列方式是有限的。

·而对于一个有限长度的位数组来说，它能表示的二进制位排列也是有限的。

根据这个原理，我们可以创建一个表，表的键为某种排列的位数组，而表的值则是相应位数组中，值为1的二进制位的数量。

创建了这种表之后，我们就可以根据输入的位数组进行查表，在无须对位数组的每个位进行检查的情况下，直接知道这个位数组包含了多少个值为1的二进制位。

举个例子，对于8位长的位数组来说，我们可以创建表格22-1，通过这个表格，我们可以一次从位数组中读入8个位，然后根据这8个位的值进行查表，直接知道这个值包含了多少个值为1的位。

表22-1　可以快速检查8位长的位数组包含多少个1

键（位数组）	值（值为 1 的位数量）
0000 0000	0
0000 0001	1
0000 0010	1
0000 0011	2
0000 0100	1
0000 0101	2
0000 0110	2
0000 0111	3
...	...
1111 1101	7
1111 1110	7
1111 1111	8

通过使用表22-1，我们只需执行一次查表操作，就可以检查8个二进制位，和之前介绍的遍历算法相比，查表法的效率提升了8倍：

·以100MB＝800000000bit（八亿位）来计算，使用查表法处理长度为100MB的位数组需要执行查表操作一亿次。

·而对于500MB长的位数组来说，使用查表法处理该位数组需要执行五亿次查表操作。

如果我们创建一个更大的表的话，那么每次查表所能处理的位就会更多，从而减少查表操作执行的次数：

·如果我们将表键的大小扩展为16位，那么每次查表就可以处理16个二进制位，检查100MB长的二进制位只需要五千万次查表，检查500MB长的二进制位只需要两亿五千万次查表。

·如果我们将表键的大小扩展为32位，那么每次查表就可以处理32个二进制位，检查100MB长的二进制位只需要两千五百万次查表，检查500MB长的二进制位只需要一亿两千五百万次查表。

初看起来，只要我们创建一个足够大的表，那么统计工作就可以轻易地完成，但这个问题实际上并没有那么简单，因为查表法的实际效果会受到内存和缓存两方面因素的限制：

·因为查表法是典型的空间换时间策略，算法在计算方面节约的时间是通过花费额外的内存换取而来的，节约的时间越多，花费的内存就越大。对于我们这里讨论的统计二进制位的问题来说，创建键长为8位的表仅需数百个字节，创建键长为16位的表也仅需数百个KB，但创建键长为32位的表却需要十多个GB。在实际中，服务器只可能接受数百个字节或者数百KB的内存消耗。

·除了内存大小的问题之外，查表法的效果还会受到CPU缓存的限制：对于固定大小的CPU缓存来说，创建的表格越大，CPU缓存所能保存的内容相比整个表格的比例就越少，查表时出现缓存不命中（cache miss）的情况就会越高，缓存的换入和换出操作就会越频繁，最终影响查表法的实际效率。

由于以上列举的两个原因，我们可以得出结论，查表法是一种比遍历算法更好的统计办法，但受限于查表法带来的内存压力，以及缓存不命中可能带来的影响，我们只能考虑创建键长为8位或者键长为16位的表，而这两种表带来的效率提升，对于处理非常长的位数组来说仍然远远不够。

为了高效地实现BITCOUNT命令，我们需要一种不会带来内存压力、并且可以在一次检查中统计多个二进制位的算法，接下来要介绍的variable-precision SWAR算法就是这样一种算法。

22.4.3　二进制位统计算法（3）：variable-precision SWAR算法

BITCOUNT命令要解决的问题——统计一个位数组中非0二进制位的数量，在数学上被称为“计算汉明重量（Hamming Weight）”。

因为汉明重量经常被用于信息论、编码理论和密码学，所以研究人员针对计算汉明重量开发了多种不同的算法，一些处理器甚至直接带有计算汉明重量的指令，而对于不具备这种特殊指令的普通处理器来说，目前已知效率最好的通用算法为variable-precision SWAR算法，该算法通过一系列位移和位运算操作，可以在常数时间内计算多个字节的汉明重量，并且不需要使用任何额外的内存。

以下是一个处理32位长度位数组的variable-precision SWAR算法的实现：

```
uint32_t swar(uint32_t i) {
    //
步骤1
```

```
    i = (i & 0x55555555) + ((i >> 1) & 0x55555555);
    //
步骤2
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
    //
步骤3
    i = (i & 0x0F0F0F0F) + ((i >> 4) & 0x0F0F0F0F);
    //
步骤4
    i = (i*(0x01010101) >> 24);
    return i;
}
```

以下是调用swar（bitarray）的执行步骤：

·步骤1计算出的值i的二进制表示可以按每两个二进制位为一组进行分组，各组的十进制表示就是该组的汉明重量。

·步骤2计算出的值i的二进制表示可以按每四个二进制位为一组进行分组，各组的十进制表示就是该组的汉明重量。

·步骤3计算出的值i的二进制表示可以按每八个二进制位为一组进行分组，各组的十进制表示就是该组的汉明重量。

·步骤4的i*0x01010101语句计算出bitarray的汉明重量并记录在二进制位的最高八位，而＞＞24语句则通过右移运算，将bitarray的汉明重量移动到最低八位，得出的结果就是bitarray的汉明重量。

举个例子，对于调用swar（0x3A70F21B），程序在第一步将计算出值0x2560A116，这个值的每两个二进制位的十进制表示记录了0x3A70F21B每两个二进制位的汉明重量，如表22-2所示。

表22-2　在对二进制进行两位分组下，0x3A70F21B的汉明重量

值	分组															
0x3A70F21B	00	11	10	10	01	11	00	00	11	11	00	10	00	01	10	11
0x2560A116	00	10	01	01	01	10	00	00	10	10	00	01	00	01	01	10
汉明重量	0	2	1	1	1	2	0	0	2	2	0	1	0	1	1	2

之后，程序在第二步将计算出值0x22304113，这个值的每四个二进制位的十进制表示记录了0x3A70F21B每四个二进制位的汉明重量，如表22-3所示。

表22-3　在对二进制进行四位分组下，0x3A70F21B的汉明重量

值	分组							
0x3A70F21B	0011	1010	0111	0000	1111	0010	0001	1011
0x22304113	0010	0010	0011	0000	0100	0001	0001	0011
汉明重量	2	2	3	0	4	1	1	3

接下来，程序在第三步将计算出值0x4030504，这个值的每八个二

进制位的十进制表示记录了0x3A70F21B每八个二进制位的汉明重量，如表22-4所示。

表22-4　在对二进制进行八位分组下，0x3A70F21B的汉明重量

值	分　组			
0x3A70F21B	00111010	01110000	11110010	00011011
0x4030504	00000100	00000011	00000101	00000100
汉明重量	4	3	5	4

在第四步，程序首先计算0x4030504*0x01010101＝0x100c0904，将汉明重量聚集到二进制位的最高八位，如表22-5所示。

表22-5　0x3A70F21B的汉明重量聚集在0x100c0904的最高八位

值	24位至31位	16至23位	8至15位	0至7位
0x100c0904	00010000	00001100	00001001	00000100
汉明重量	16	无用值	无用值	无用值

之后程序计算0x100c0904 ＞＞ 24，将汉明重量移动到低八位，最终得出值0x10，也即是十进制值16，这个值就是0x3A70F21B的汉明重量，如表22-6所示。

表22-6　进行移位之后，0x3A70F21B的汉明重量

值	24位至31位	16至23位	8至15位	0至7位
0x10	00000000	00000000	00000000	00010000

swar函数每次执行可以计算32个二进制位的汉明重量，它比之前介绍的遍历算法要快32倍，比键长为8位的查表法快4倍，比键长为16位的查表法快2倍，并且因为swar函数是单纯的计算操作，所以它无须像查表法那样，使用额外的内存。

另外，因为swar函数是一个常数复杂度的操作，所以我们可以按照自己的需要，在一次循环中多次执行swar，从而按倍数提升计算汉明重量的效率：

·例如，如果我们在一次循环中调用两次swar函数，那么计算汉明重量的效率就从之前的一次循环计算32位提升到了一次循环计算64位。

·又例如，如果我们在一次循环中调用四次swar函数，那么一次循环就可以计算128个二进制位的汉明重量，这比每次循环只调用一次swar函数要快四倍！

当然，在一个循环里执行多个swar调用这种优化方式是有极限的：一旦循环中处理的位数组的大小超过了缓存的大小，这种优化的效果就会降低并最终消失。

22.4.4 二进制位统计算法（4）：Redis的实现

BITCOUNT命令的实现用到了查表和variable-precisionSWAR两种算法：

·查表算法使用键长为8位的表，表中记录了从0000 0000到1111 1111在内的所有二进制位的汉明重量。

·至于variable-precision SWAR算法方面，BITCOUNT命令在每次循环中载入128个二进制位，然后调用四次32位variable-precision SWAR算法来计算这128个二进制位的汉明重量。

在执行BITCOUNT命令时，程序会根据未处理的二进制位的数量来决定使用那种算法：

·如果未处理的二进制位的数量大于等于128位，那么程序使用variable-precision SWAR算法来计算二进制位的汉明重量。

·如果未处理的二进制位的数量小于128位，那么程序使用查表算法来计算二进制位的汉明重量。

以下伪代码展示了BITCOUNT命令的实现原理：

```
#
一个表，记录了所有八位长位数组的汉明重量
#
程序将8
位长的位数组转换成无符号整数，并在表中进行索引
#
例如，对于输入0000 0011
，程序将二进制转换为无符号整数3
#
然后取出weight_in_byte[3]
的值2
# 2
就是0000 0011
的汉明重量
weight_in_byte = [0,1,1,2,1,2,2,/*...*/,7,7,8]
def BITCOUNT(bits):
    #
计算位数组包含了多少个二进制位
    count = count_bit(bits)
    #
初始化汉明重量为零
    weight = 0
    #
如果未处理的二进制位大于等于128
位
    #
那么使用variable-precision SWAR
算法来处理
    while count >= 128:
        #
四个swar
调用，每个调用计算32
个二进制位的汉明重量
        #
注意：bits[i:j]
```

```
中的索引j
是不包含在取值范围之内的
        weight += swar(bits[0:32])
        weight += swar(bits[32:64])
        weight += swar(bits[64:96])
        weight += swar(bits[96:128])
        #
移动指针，略过已处理的位，指向未处理的位
        bits = bits[128:]
        #
减少未处理位的长度
        count -= 128
    #
如果执行到这里，说明未处理的位数量不足128
位
    #
那么使用查表法来计算汉明重量
    while count:
        #
将8
个位转换成无符号整数，作为查表的索引（键）
        index = bits_to_unsigned_int(bits[0:8])
        weight += weight_in_byte[index]
        #
移动指针，略过已处理的位，指向未处理的位
        bits = bits[8:]
        #
减少未处理位的长度
        count -= 8
    #
计算完毕，返回输入二进制位的汉明重量
    return weight
```

这个BITCOUNT实现的算法复杂度为O（n），其中n为输入二进制位的数量。

更具体一点，我们可以用以下公式来计算BITCOUNT命令在处理长度为n的二进制位输入时，命令中的两个循环需要执行的次数：

·第一个循环的执行次数可以用公式$loop_1 = n \div 128$」计算得出。

·第二个循环的执行次数可以用公式$loop_2 = n \bmod 128$计算得出。

以100MB＝800000000bit来计算，BITCOUNT命令处理一个100MB长的位数组共需要执行第一个循环六百二十五万次，第二个循环零次。以500MB＝4000000000bit来计算，BITCOUNT命令处理一个500MB长的位数组共需要执行第一个循环三千一百二十五万次，第二个循环零次。

通过使用更好的算法，我们将计算100MB和500MB长的二进制位所需的循环次数从最开始使用遍历算法时的数亿甚至数十亿次减少到了数百万次和数千万次。

22.5　BITOP命令的实现

因为C语言直接支持对字节执行逻辑与（&）、逻辑或（|）、逻辑异或（^）和逻辑非（~）操作，所以BITOP命令的AND、OR、XOR和NOT四个操作都是直接基于这些逻辑操作实现的：

·在执行BITOP　AND命令时，程序用&操作计算出所有输入二进制位的逻辑与结果，然后保存在指定的键上面。

·在执行BITOP　OR命令时，程序用|操作计算出所有输入二进制位的逻辑或结果，然后保存在指定的键上面。

·在执行BITOP　XOR命令时，程序用^操作计算出所有输入二进制位的逻辑异或结果，然后保存在指定的键上面。

·在执行BITOP　NOT命令时，程序用~操作计算出输入二进制位的逻辑非结果，然后保存在指定的键上面。

举个例子，假设客户端执行命令：

```
BITOP AND result x y
```

其中，键x保存的位数组如图22-18所示，而键y保存的位数组如图22-19所示，BITOP命令将执行以下操作：

buf[0]	1	0	1	0	0	1	0	1
buf[1]	1	1	0	0	0	0	1	1
buf[2]	0	0	0	0	1	1	1	1
buf[3]	0	0	0	0	0	0	0	0

图22-18　键x所保存的位数组

buf[0]	1	1	1	1	1	1	1	1
buf[1]	0	0	0	0	0	0	0	0
buf[2]	1	1	1	1	1	1	1	1
buf[3]	0	0	0	0	0	0	0	0

图22-19　键y所保存的位数组

1）创建一个空白的位数组value，用于保存AND操作的结果。

2）对两个位数组的第一个字节执行buf[0] & buf[0]操作，并将结果保存到value[0]字节。

3）对两个位数组的第二个字节执行buf[1] & buf[1]操作，并将结果保存到value[1]字节。

4）对两个位数组的第三个字节执行buf[2] & buf[2]操作，并将结果保存到value[2]字节。

5）经过前面的三次逻辑与操作，程序得到了图22-20所示的计算结果，并将它保存在键result上面。

buf[0]	1	0	1	0	0	1	0	1
buf[1]	0	0	0	0	0	0	0	0
buf[2]	0	0	0	0	1	1	1	1
buf[3]	0	0	0	0	0	0	0	0

图22-20　键x和键y执行BITOP AND命令产生的结果

BITOP OR、BITOP XOR、BITOP NOT命令的执行过程和这里列出的BITOP AND的执行过程类似。

因为BITOP AND、BITOP OR、BITOP XOR三个命令可以接受多个位数组作为输入，程序需要遍历输入的每个位数组的每个字节来进行计算，所以这些命令的复杂度为O（n2）；与此相反，因为BITOP NOT命令只接受一个位数组输入，所以它的复杂度为O（n）。

22.6 重点回顾

·Redis使用SDS来保存位数组。

·SDS使用逆序来保存位数组，这种保存顺序简化了SETBIT命令的实现，使得SETBIT命令可以在不移动现有二进制位的情况下，对位数组进行空间扩展。

·BITCOUNT命令使用了查表算法和variable-precision SWAR算法来优化命令的执行效率。

·BITOP命令的所有操作都使用C语言内置的位操作来实现。

22.7 参考资料

·StackOverflow网站上的一个帖子对Hamming Weight主题进行了讨论，并给出了有用的参考信息：http://stackoverflow.com/questions/109023/how-to-count-the-number-of-set-bits-in-a-32-bit-integer。

·博客文章《Counting The Number Of Set Bits In An Integer》给出了variable-precision SWAR算法的介绍：http://yesteapea.wordpress.com/2013/03/03/counting-the-number-of-set-bits-in-an-integer/。

第23章　慢查询日志

Redis的慢查询日志功能用于记录执行时间超过给定时长的命令请求，用户可以通过这个功能产生的日志来监视和优化查询速度。

服务器配置有两个和慢查询日志相关的选项：

·slowlog-log-slower-than选项指定执行时间超过多少微秒（1秒等于1 000 000微秒）的命令请求会被记录到日志上。

举个例子，如果这个选项的值为100，那么执行时间超过100微秒的命令就会被记录到慢查询日志；如果这个选项的值为500，那么执行时间超过500微秒的命令就会被记录到慢查询日志。

·slowlog-max-len选项指定服务器最多保存多少条慢查询日志。

服务器使用先进先出的方式保存多条慢查询日志，当服务器存储的慢查询日志数量等于slowlog-max-len选项的值时，服务器在添加一条新的慢查询日志之前，会先将最旧的一条慢查询日志删除。

举个例子，如果服务器slowlog-max-len的值为100，并且假设服务器已经储存了100条慢查询日志，那么如果服务器打算添加一条新日志的话，它就必须先删除目前保存的最旧的那条日志，然后再添加新日志。

我们来看一个慢查询日志功能的例子，首先用CONFIG　SET命令将slowlog-log-slower-than选项的值设为0微秒，这样Redis服务器执行的任何命令都会被记录到慢查询日志中，接着将slowlog-max-len选项的值设为5，让服务器最多只保存5条慢查询日志：

```
redis> CONFIG SET slowlog-log-slower-than 0
OK
redis> CONFIG SET slowlog-max-len 5
OK
```

接着，我们用客户端发送几条命令请求：

```
redis> SET msg "hello world"
OK
redis> SET number 10086
OK
redis> SET database "Redis"
OK
```

然后使用SLOWLOG GET命令查看服务器所保存的慢查询日志：

```
redis> SLOWLOG GET
1) 1)
```

```
(integer
) 4                  #
日志的唯一标识符(uid
)
   2) (integer) 1378781447      #
命令执行时的UNIX
时间戳
   3) (integer) 13              #
命令执行的时长，以微秒计算
   4) 1) "SET"                  #
命令以及命令参数
      2) "database"
      3) "Redis"
2) 1) (integer) 3
   2) (integer) 1378781439
   3) (integer) 10
   4) 1) "SET"
      2) "number"
      3) "10086"
3) 1) (integer) 2
   2) (integer) 1378781436
   3) (integer) 18
   4) 1) "SET"
      2) "msg"
      3) "hello world"
4) 1) (integer) 1
   2) (integer) 1378781425
   3) (integer) 11
   4) 1) "CONFIG"
      2) "SET"
      3) "slowlog-max-len"
      4) "5"
5) 1) (integer) 0
   2) (integer) 1378781415
   3) (integer) 53
   4) 1) "CONFIG"
      2) "SET"
      3) "slowlog-log-slower-than"
      4) "0"
```

如果这时再执行一条SLOWLOG GET命令，那么我们将看到，上一次执行的SLOWLOG GET命令已经被记录到了慢查询日志中，而最旧的、ID为0的慢查询日志已经被删除，服务器的慢查询日志数量仍然为5条：

```
redis> SLOWLOG GET
1) 1) (integer) 5
   2) (integer) 1378781521
   3) (integer) 61
   4) 1) "SLOWLOG"
      2) "GET"
2) 1) (integer) 4
   2) (integer) 1378781447
   3) (integer) 13
   4) 1) "SET"
      2) "database"
      3) "Redis"
3) 1) (integer) 3
   2) (integer) 1378781439
   3) (integer) 10
   4) 1) "SET"
      2) "number"
      3) "10086"
4) 1) (integer) 2
   2) (integer) 1378781436
   3) (integer) 18
   4) 1) "SET"
```

```
      2) "msg"
      3) "hello world"
5) 1) (integer) 1
   2) (integer) 1378781425
   3) (integer) 11
   4) 1) "CONFIG"
      2) "SET"
      3) "slowlog-max-len"
      4) "5"
```

23.1　慢查询记录的保存

服务器状态中包含了几个和慢查询日志功能有关的属性：

```
struct redisServer {
    // ...
    //
下一条慢查询日志的ID
    long long slowlog_entry_id;
    //
保存了所有慢查询日志的链表
    list *slowlog;
    //
服务器配置slowlog-log-slower-than
选项的值
    long long slowlog_log_slower_than;
    //
服务器配置slowlog-max-len
选项的值
    unsigned long slowlog_max_len;
    // ...
};
```

slowlog_entry_id属性的初始值为0，每当创建一条新的慢查询日志时，这个属性的值就会用作新日志的id值，之后程序会对这个属性的值增一。

例如，在创建第一条慢查询日志时，slowlog_entry_id的值0会成为第一条慢查询日志的ID，而之后服务器会对这个属性的值增一；当服务器再创建新的慢查询日志的时候，slowlog_entry_id的值1就会成为第二条慢查询日志的ID，然后服务器再次对这个属性的值增一，以此类推。

slowlog链表保存了服务器中的所有慢查询日志，链表中的每个节点都保存了一个slowlogEntry结构，每个slowlogEntry结构代表一条慢查询日志：

```
typedef struct slowlogEntry {
    //
唯一标识符
    long long id;
    //
命令执行时的时间，格式为UNIX
时间戳
    time_t time;
    //
执行命令消耗的时间，以微秒为单位
    long long duration;
    //
命令与命令参数
    robj **argv;
    //
命令与命令参数的数量
    int argc;
} slowlogEntry;
```

举个例子，对于以下慢查询日志来说：

```
1) (integer) 3
2) (integer) 1378781439
3) (integer) 10
4) 1) "SET"
   2) "number"
   3) "10086"
```

图23-1展示的就是该日志所对应的slowlogEntry结构。

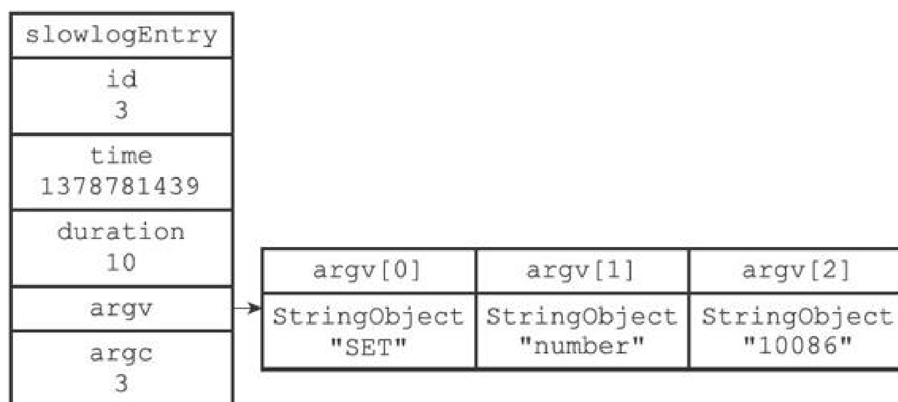


图23-1　slowlogEntry结构示例

图23-2展示了服务器状态中和慢查询功能有关的属性：

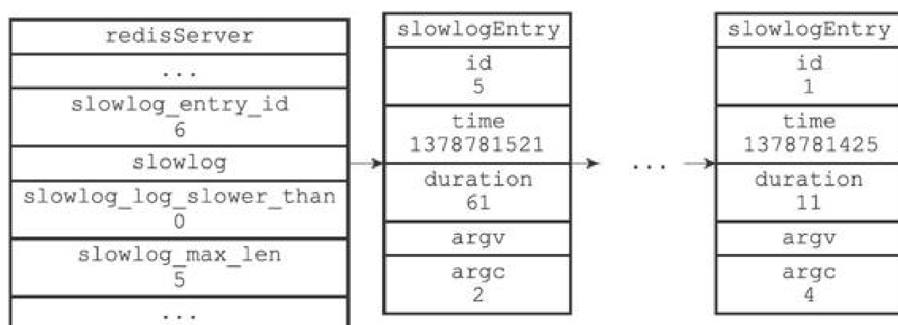


图23-2　redisServer结构示例

·slowlog_entry_id的值为6，表示服务器下条慢查询日志的id值将为6。

·slowlog链表包含了id为5至1的慢查询日志，最新的5号日志排在链表的表头，而最旧的1号日志排在链表的表尾，这表明slowlog链表是使用插入到表头的方式来添加新日志的。

·slowlog_log_slower_than记录了服务器配置slowlog-log-slower-than选项的值0，表示任何执行时间超过0微秒的命令都会被慢查询日志记录。

·slowlog-max-len属性记录了服务器配置slowlog-max-len选项的值5，表示服务器最多储存五条慢查询日志。

注意

因为版面空间不足，所以图23-2展示的各个slowlogEntry结构都省略了argv数组。

23.2　慢查询日志的阅览和删除

弄清楚了服务器状态的slowlog链表的作用之后，我们可以用以下伪代码来定义查看日志的SLOWLOG GET命令：

```
def SLOWLOG_GET(number=None):
    #
用户没有给定number
参数
    #
那么打印服务器包含的全部慢查询日志
    if number is None:
        number = SLOWLOG_LEN()
    #
遍历服务器中的慢查询日志
    for log in redisServer.slowlog:
        if number <= 0:
            #
打印的日志数量已经足够，跳出循环
            break
        else:
            #
继续打印，将计数器的值减一
            number -= 1
        #
打印日志
        printLog(log)
```

查看日志数量的SLOWLOG LEN命令可以用以下伪代码来定义：

```
def SLOWLOG_LEN():
    # slowlog
链表的长度就是慢查询日志的条目数量
    return len(redisServer.slowlog)
```

另外，用于清除所有慢查询日志的SLOWLOG　RESET命令可以用以下伪代码来定义：

```
def SLOWLOG_RESET():
    #
遍历服务器中的所有慢查询日志
    for log in redisServer.slowlog:
        #
删除日志
        deleteLog(log)
```

23.3 添加新日志

在每次执行命令的之前和之后，程序都会记录微秒格式的当前UNIX时间戳，这两个时间戳之间的差就是服务器执行命令所耗费的时长，服务器会将这个时长作为参数之一传给slowlogPushEntryIfNeeded函数，而slowlogPushEntryIfNeeded函数则负责检查是否需要为这次执行的命令创建慢查询日志，以下伪代码展示了这一过程：

```
# 
记录执行命令前的时间
before = unixtime_now_in_us()
# 
执行命令
execute_command(argv, argc, client)
# 
记录执行命令后的时间
after = unixtime_now_in_us()
# 
检查是否需要创建新的慢查询日志
slowlogPushEntryIfNeeded(argv, argc, before-after)
```

slowlogPushEntryIfNeeded函数的作用有两个：

1）检查命令的执行时长是否超过slowlog-log-slower-than选项所设置的时间，如果是的话，就为命令创建一个新的日志，并将新日志添加到slowlog链表的表头。

2）检查慢查询日志的长度是否超过slowlog-max-len选项所设置的长度，如果是的话，那么将多出来的日志从slowlog链表中删除掉。

以下是slowlogPushEntryIfNeeded函数的实现代码：

```
void slowlogPushEntryIfNeeded(robj **argv, int argc, long long duration) {
    // 
慢查询功能未开启，直接返回
    if (server.slowlog_log_slower_than < 0) return;
    // 
如果执行时间超过服务器设置的上限，那么将命令添加到慢查询日志
    if (duration >= server.slowlog_log_slower_than)
        // 
新日志添加到链表表头
        listAddNodeHead(server.slowlog,slowlogCreateEntry(argv,argc,duration));
    // 
如果日志数量过多，那么进行删除
    while (listLength(server.slowlog) > server.slowlog_max_len)
        listDelNode(server.slowlog,listLast(server.slowlog));
}
```

函数中的大部分代码我们已经介绍过了，唯一需要说明的是slowlogCreateEntry函数：该函数根据传入的参数，创建一个新的慢查询日志，并将redisServer.slowlog_entry_id的值增1。

举个例子，假设服务器当前保存的慢查询日志如图23-2所示，如果

我们执行以下命令：

```
redis> EXPIRE msg 10086
(integer) 1
```

服务器在执行完这个EXPIRE命令之后，就会调用slowlogPushEntryIfNeeded函数，函数将为EXPIRE命令创建一条id为6的慢查询日志，并将这条新日志添加到slowlog链表的表头，如图23-3所示。

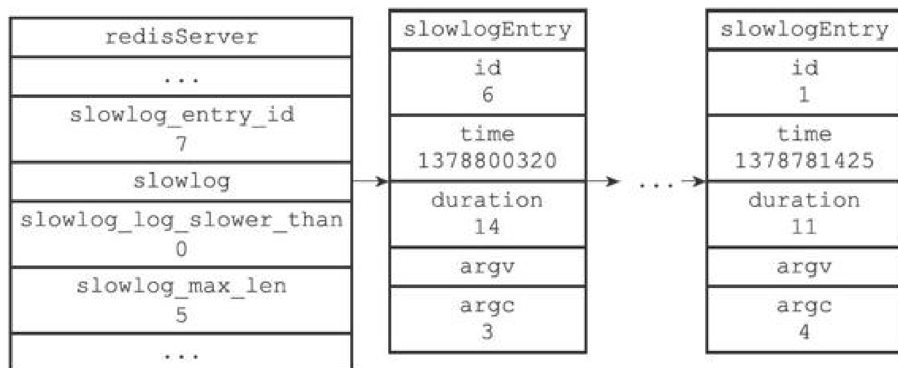


图23-3　EXPIRE命令执行之后的服务器状态

注意，除了slowlog链表发生了变化之外，slowlog_entry_id的值也从6变为7了。

之后，slowlogPushEntryIfNeeded函数发现，服务器设定的最大慢查询日志数目为5条，而服务器目前保存的慢查询日志数目为6条，于是服务器将id为1的慢查询日志删除，让服务器的慢查询日志数量回到设定好的5条。

删除操作执行之后的服务器状态如图23-4所示。

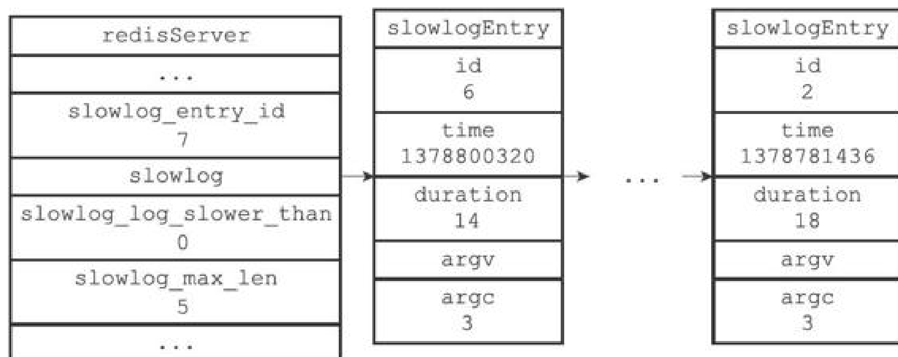


图23-4　删除id为1的慢查询日志之后的服务器状态

23.4 重点回顾

·Redis的慢查询日志功能用于记录执行时间超过指定时长的命令。

·Redis服务器将所有的慢查询日志保存在服务器状态的slowlog链表中，每个链表节点都包含一个slowlogEntry结构，每个slowlogEntry结构代表一条慢查询日志。

·打印和删除慢查询日志可以通过遍历slowlog链表来完成。

·slowlog链表的长度就是服务器所保存慢查询日志的数量。

·新的慢查询日志会被添加到slowlog链表的表头，如果日志的数量超过slowlog-max-len选项的值，那么多出来的日志会被删除。

第24章　监视器

通过执行MONITOR命令，客户端可以将自己变为一个监视器，实时地接收并打印出服务器当前处理的命令请求的相关信息：

```
redis> MONITOR
OK
1378822099.421623 [0 127.0.0.1:56604] "PING"
1378822105.089572 [0 127.0.0.1:56604] "SET" "msg" "hello world"
1378822109.036925 [0 127.0.0.1:56604] "SET" "number" "123"
1378822140.649496 [0 127.0.0.1:56604] "SADD" "fruits" "Apple" "Banana" "Cherry"
1378822154.117160 [0 127.0.0.1:56604] "EXPIRE" "msg" "10086"
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
1378822258.690131 [0 127.0.0.1:56604] "DBSIZE"
```

每当一个客户端向服务器发送一条命令请求时，服务器除了会处理这条命令请求之外，还会将关于这条命令请求的信息发送给所有监视器，如图24-1所示。

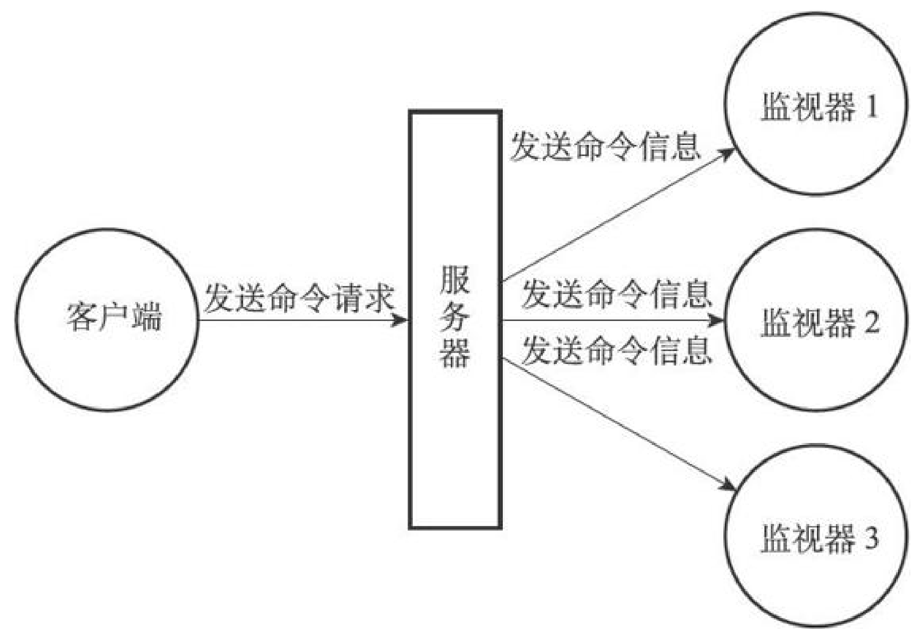


图24-1　命令的接收和信息的发送

24.1 成为监视器

发送MONITOR命令可以让一个普通客户端变为一个监视器，该命令的实现原理可以用以下伪代码来实现：

```
def MONITOR():
    #
打开客户端的监视器标志
    client.flags |= REDIS_MONITOR
    #
将客户端添加到服务器状态的monitors
链表的末尾
    server.monitors.append(client)
    #
向客户端返回OK
    send_reply("OK")
```

举个例子，如果客户端c10086向服务器发送MONITOR命令，那么这个客户端的REDIS_MONITOR标志会被打开，并且这个客户端本身会被添加到monitors链表的表尾。

假设客户端c10086发送MONITOR命令之前，monitors链表的状态如图24-2所示，那么在服务器执行客户端c10086发送的MONITOR命令之后，monitors链表将被更新为图24-3所示的状态。

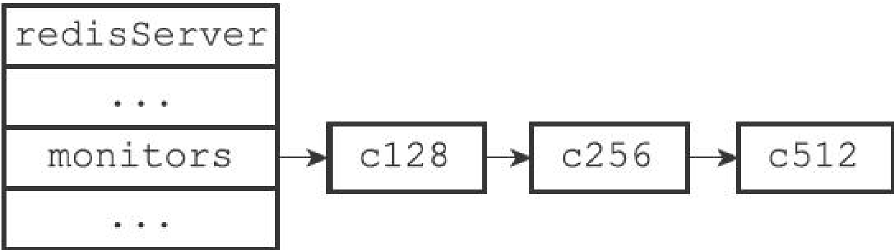


图24-2　客户端c10086执行MONITOR命令之前的monitors链表

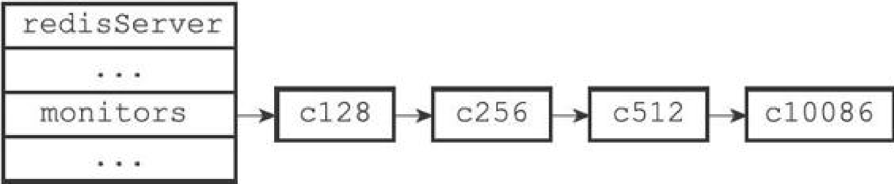


图24-3　客户端c10086执行MONITOR命令之后的monitors链表

24.2 向监视器发送命令信息

服务器在每次处理命令请求之前，都会调用replicationFeedMonitors函数，由这个函数将被处理的命令请求的相关信息发送给各个监视器。

以下是replicationFeedMonitors函数的伪代码定义，函数首先根据传入的参数创建信息，然后将信息发送给所有监视器：

```
def replicationFeedMonitors(client, monitors, dbid, argv, argc):
    #
根据执行命令的客户端、当前数据库的号码、命令参数、命令参数个数等参数
    #
创建要发送给各个监视器的信息
    msg = create_message(client, dbid, argv, argc)
    #
遍历所有监视器
    for monitor in monitors:
        #
将信息发送给监视器
        send_message(monitor, msg)
```

举个例子，假设服务器在时间1378822257.329412，根据IP为127.0.0.1、端口号为56604的客户端发送的命令请求，对0号数据库执行命令KEYS*，那么服务器将创建以下信息：

```
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
```

如果服务器monitors链表的当前状态如图24-3所示，那么服务器会分别将信息发送给c128、c256、c512和c10086四个监视器，如图24-4所示。

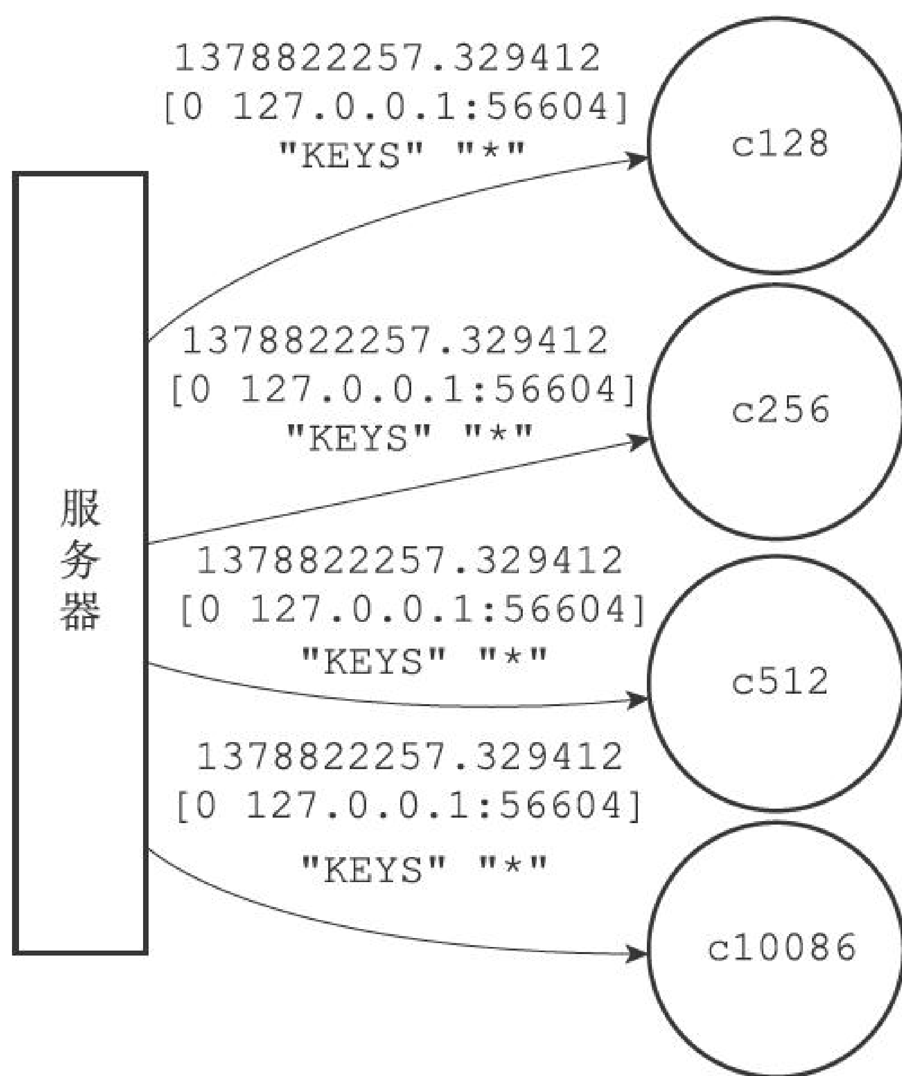


图24-4　服务器将信息发送给各个监视器

24.3 重点回顾

·客户端可以通过执行MONITOR命令，将客户端转换成监视器，接收并打印服务器处理的每个命令请求的相关信息。

·当一个客户端从普通客户端变为监视器时，该客户端的REDIS_MONITOR标识会被打开。

·服务器将所有监视器都记录在monitors链表中。

·每次处理命令请求时，服务器都会遍历monitors链表，将相关信息发送给监视器。